University of Zurich[UZH]

Department of Informatics

Domain-Specific Scheduling Protocols

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zurich

for the degree of
Doctor of Science (Ph.D.)

by

Christian Tilgner

from Germany

Accepted on the recommendation of

Prof. Dr. Michael Böhlen
Dr. Carl-Christian Kanne

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, 24.10.2012

The head of the Ph.D. program in informatics: Prof. Abraham Bernstein, Ph.D.

Abstract

A database scheduler takes requests from transactions and generates a request order that fulfills the constraints of a scheduling protocol, e.g., correctness criteria. The goal of this thesis is to provide a new method for the development of domain-specific protocols for scheduling database requests.

Scheduling concurrent requests is an ubiquitous problem in modern server systems based on, e.g., Web Services that handle large numbers of concurrent requests. These systems require user scalability, performance predictability, and flexibility, i.e., the ability to adapt to domain-specific needs, e.g., relaxed correctness criteria or service-level agreements (SLAs). The traditional approach of outsourcing scheduling to database management systems (DBMSs) is of limited applicability for these systems, because DBMSs provide only a limited amount of predefined consistency levels and limited user scalability. The state of the art is to develop application-specific schedulers for a given application from scratch which yields fine-tuned schedulers satisfying the application's scheduling constraints, albeit at a great cost and with long development times. Imperative implementations of schedulers can be complex, hard to verify, and adapting such schedulers results in expensive and error-prone re-implementations.

The solution we propose for the development of domain-specific scheduling protocols is a generic scheduling model called Oshiya. The main idea of this model is to treat requests as data and employ database query processing techniques to produce request schedules. Oshiya can express traditional and domain-specific scheduling protocols. We introduce an Oshiya implementa-

tion of the traditional strong two-phase locking protocol and leverage the conciseness of Oshiya protocol implementations to prove its correctness. Our experiments show that for large numbers of concurrent requests our approach provides a better performance than a native database scheduler. Oshiya protocol implementations can be adapted easily to modified scheduling constraints. We leverage this advantage and develop the Declarative Serializable Snapshot Isolation protocol, a modified version of the Snapshot Isolation protocol, and prove that it produces serializable histories. We propose the resource acquisition protocol (RAP), a domain-specific protocol for scheduling transactions that compete for resources that are available in limited quantity, which is a typical usage pattern in booking, reservation, and web shop systems. We prove that RAP is deadlock-free and that it produces less aborts due to insufficient resource availability than SI. Our experimental results confirm that RAP performs better than SS2PL and SI with respect to aborts and throughput.

We present the Oshiya Debugger and Analyzer (ODA), a novel system for debugging, visualizing, and comparing scheduling protocols developed using Oshiya. ODA supports the simultaneous execution of single- and multiversion protocols, breakpoints, backward and forward debugging, as well as the statistical and visual protocol analysis.

Acknowledgments

I would like to express my sincerest gratitude to all people who have inspired, motivated and supported me during my PhD study.

First, I would like to thank my advisors Michael Böhlen and Carl-Christian Kanne. I am deeply grateful for their support, their valuable advice, spending so much time on discussions about my research and iterating over the papers, and that I could profit from their knowledge and experience. Thank you for being willing to take over the supervision.

I thank Klaus R. Dittrich who passed away in late 2007 for introducing me to research and giving me the opportunity to pursue my PhD study at the University of Zurich.

I also would like to thank my colleagues from the Database Technology Group at the University of Zurich for their help whenever help was needed, their support, and all the memorable social activities.

I thank my co-authors Michael Böhlen, Carl-Christian Kanne, Klaus R. Dittrich, Dietrich Christopeit, Boris Glavic, Patrick Leibundgut, Luis Schüller, and Patrick Ziegler.

Special thanks to Boris Glavic for his tirelessly help, motivation, support, and for all the many helpful discussions.

Finally, I would like to thank my family, my extended family and all my friends who always supported me. In particular I would like to thank my wife Sarah for her love and support. She

always believed in me and encouraged me regardless of all the time that we could not spend together due to my research.

Christian Tilgner
Zurich, July 2012

To my family

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Domain-Specific Scheduling Protocols

A database scheduler takes requests from transactions and generates a request order that fulfills the constraints of a scheduling protocol, e.g., correctness criteria. The goal of this thesis is to provide a new method for developing domain-specific protocols for the scheduling of database requests.

Modern server systems based on, e.g., Web Services handle large numbers of concurrent requests which have to be scheduled according to constraints including, e.g., correctness criteria or service-level agreements (SLAs). These systems require user scalability, performance predictability, and flexibility, i.e., the ability to adapt to domain-specific needs.

The traditional approach of outsourcing scheduling to database management systems (DBMSs) is of limited applicability for these systems, because DBMSs only provide a limited user scalability and predictability of performance if many concurrent clients access the same database. DBMSs are inflexible because they only offer a limited amount of predefined consistency levels and do not provide sophisticated support for SLAs such as an effective differentiation between

transactions, e.g., transactions from users with different priorities [Sch06a]. This is why they are not perfectly applicable for highly scalable systems and why they often cannot be used to satisfy domain-specific scheduling requirements [DHJ+07, FK09].

In practice, it turned out that relaxed consistency is necessary for highly scalable systems [FBJ09]. Techniques like strict or strong consistency and database-style transactions do not scale at Internet level and are rarely needed in modern large-scale systems [BFG+08, Vog07, HC09]. For most parts of modern highly scalable web applications, e.g., hotel or flight reservation systems, or Internet shops like Amazon relaxed consistency is sufficient. Thus, there is a need for new consistency levels which are different from the SQL isolation levels used by DBMSs [FK09, Vog07].

The state of the art is to develop application-specific schedulers imperatively for a given application, e.g., Amazon, Ebay, or Yahoo [CRS+08, Vog07]. The advantage of this approach is that it yields fine-tuned schedulers satisfying the application's scheduling constraints (consistency etc.), albeit at a great cost and with long development times. Imperative implementations of schedulers can be complex, hard to verify, and difficult to understand, especially if the correctness criteria are less well studied than, e.g., classical serializability. Given the rapid evolution of web applications, a critical issue for the development of schedulers is developer productivity. Applications using imperatively implemented algorithms are not flexible enough to react to changing requirements or evolving business processes. Adapting such schedulers to rapidly evolving requirements which affect the correctness criteria and SLAs results in expensive and error-prone re-implementations.

In the next section, we show an application area for domain-specific scheduling protocols. Our solutions are described in Section 1.3. Section 1.4 discusses the organization of the thesis.

1.2 Application Area

In this section, we present an application which we use to illustrate the shortcomings of traditional scheduling protocols. The illustrated flight booking scenario can be viewed as an instance of the general problem of scheduling concurrent requests that compete for resources, whereby, each resource is available in limited quantity. There are numerous examples from the e-commerce domain where this pattern can be observed, e.g., booking systems, reservations systems, and web shops.

Consider a flight database containing flights between Paris and Berlin. The flights are recorded in relation S, shown in Table 1.1, where *ID* is the unique flight identifier, *Seat* is the quantity of available seats for this flight, *Dep* is the departure airport, *Dest* is the destination airport, *TD* is the departure time, *TA* is the arrival time, *Dur* is the flight duration, *Stp* is the number of stopovers and *Prc* the price of the flight. For instance, tuple s_1 records non-stop flight u from Berlin to Paris with a duration of 1h 45m for a price of 210 with 15 available seats.

	S								
	ID	**Seat**	**Dep**	**Dest**	**TD**	**TA**	**Dur**	**Stp**	**Prc**
s_1	u	15	Berlin	Paris	07:00	08:45	01:45	0	210
s_2	v	99	Paris	Berlin	09:00	10:35	01:35	0	310
s_3	w	67	Berlin	Paris	11:00	12:45	01:45	0	150
s_4	x	7	Paris	Berlin	13:00	14:35	01:35	0	300
s_5	y	15	Berlin	Paris	18:00	21:45	03:45	1	255
s_6	z	34	Paris	Berlin	22:00	01:30	03:30	1	210

Table 1.1: Sample Flight Database

Users Alice and Bob search for flights. Alice browses (reads) flights x and y and books both of them, each with a quantity of 2. Bob browses flight y and books it with a quantity of 2. Their requests are listed below; $r(x)$ checks the availability of a resource (a read request) and $w(x)$ acquires a resource (a write request).

$$\begin{aligned} \text{Bob:} \quad & r(y)\ w(y) \\ \text{Alice:} \quad & r(x)\ w(x)\ r(y)\ w(y) \end{aligned}$$

Applying generic scheduling protocols to such a scenario is problematic. Generic protocols are oblivious to the semantics of the requests which leads to unnecessary blocking and aborts. For instance, if the strong two-phase locking protocol (SS2PL) is applied to generate a request order (schedule) for the requests of Alice and Bob, their concurrent execution can lead to a deadlock and only one can proceed. This is the case if Alice and Bob both acquired shared locks on y and then request exclusive locks on y in order to write y, as shown in Figure 1.1. Dotted lines denote blocking and blocked requests are added in angle brackets. The blocking is not strictly necessary, because there are enough seats available on this flight to apply the requests of Alice and Bob in any order without violating consistency.

If Snapshot Isolation (SI) is applied to schedule the concurrent transactions of Alice and Bob, then only one can commit. This is because Snapshot Isolation requires that concurrently executed committed transactions did not write a same object. This situation is illustrated in Figure 1.2.

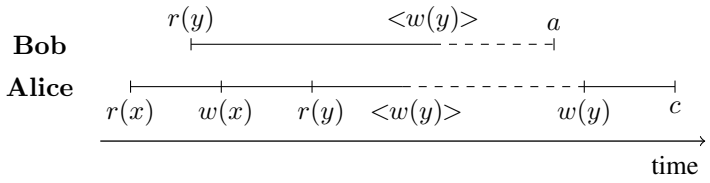


Figure 1.1: SS2PL Schedule with Deadlock

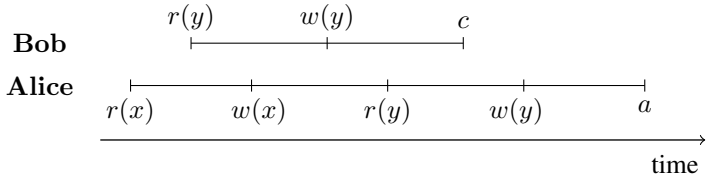


Figure 1.2: Snapshot Isolation Schedule with Abort

A protocol that is specifically tailored for this application domain and that considers the semantics of the requests can significantly improve performance and user experience (less aborts).

1.3 Contributions

Our solution for the development of domain-specific protocols for scheduling database requests is the Oshiya scheduling model. Oshiya models pending and historical requests as data collections and uses database query processing techniques to generate request schedules. We introduce the declarative two-phase locking protocol (DSS2PL), an Oshiya implementation of the traditional strong two-phase locking protocol. We propose Declarative Serializable Snapshot Isolation (DSSI), a serializable version of Snapshot Isolation. We leverage the conciseness of Oshiya protocol implementations and prove their correctness. We develop the resource acquisition protocol (RAP), a domain-specific protocol for scheduling reservation resp. shopping transactions like the transactions of Alice and Bob described in the previous subsection. Experiments show that the declarative approach can compete with a highly optimized database scheduler for large numbers of concurrent requests. Furthermore, we present the Oshiya Debugger and Analyzer (ODA), a tool for debugging, visualizing, and comparing scheduling protocols developed using the Oshiya scheduling model.

1.3.1 Oshiya

Oshiya[1] is a declarative scheduling model that facilitates the definition of domain-specific protocols for scheduling database requests. The main idea is to treat sets of requests as data collections and to employ database query processing techniques over this request data to produce request schedules in an efficient and flexible manner. Scheduling protocols are implemented as queries, the so-called scheduling queries, that select those requests that can be executed next without violating the scheduling constraints. This approach has several advantages: (1) Oshiya is powerful enough to model a wide range of scheduling protocols from traditional lock-based protocols to specialized protocols that make rigorous use of application semantics. Furthermore, Oshiya provides support for SLAs such as a differentiation between transactions, e.g., transactions from users with different priorities. We developed scheduling queries for the strong two-phase locking (SS2PL) protocol, a serializable version of Snapshot Isolation, and for a relaxed consistency protocol. (2) Oshiya allows to specify scheduling protocols close to their formal definition which facilitates reasoning over properties of protocol implementations such as verifying their correctness. We have proven the correctness of the scheduling queries implementing SS2PL. (3) Instead of scheduling each request on its own, Oshiya schedules multiple requests at once, amortizing scheduling costs. Our experiments show that the declarative scheduling approach can perform better than a regular scheduler for high client counts. (4) When implementing a scheduler, developers can focus on the protocol implementation (the scheduling queries) itself, which decreases the amount of code and the effort needed to implement or adapt schedulers. The declarative formulation of scheduling protocols is much more concise, easier to understand and easier to modify than an imperative scheduler implementation. (5) The Oshiya model purposely ignores lower level aspects of scheduling that are independent of the scheduling constraints such as queueing of incoming requests, executing requests, communication between the scheduler and the clients, or managing (network) connections. Their implementation is in the responsibility of the system that implements Oshiya. For the experiments, we use a middleware implementation of Oshiya called Smile which takes care of these low level aspects.

[1] Oshiya refers to the passenger arrangement staff at Japanese train stations who help to fill a train by pushing people onto the train or guiding people to free railway cars.

1.3.2 Declarative Strong Two-Phase Locking Protocol

We illustrate Oshiya by developing the *declarative two-phase locking protocol* (DSS2PL), an Oshiya implementation of the traditional strong (rigorous) two-phase locking protocol (SS2PL). We formalize the DSS2PL protocol constraints as Oshiya protocol specification formulated as first-order predicate logic expressions and develop an SQL implementation of DSS2PL using Oshiya. We prove that the DSS2PL implementation guarantees histories that conform to the protocol specification. We evaluate DSS2PL using Smile, an Oshiya middleware implementation that allows to plug in scheduling policies in form of queries, and to compare their execution to native schedulers. We conduct experiments that compare the performance and the predictability of DSS2PL with a native database scheduler. Our experiments show that for large client counts our implementation performs better than the native database scheduler. Our approach scales better than the native scheduler and provides higher stability and, thus, predictable performance. This is an advantage of scheduling multiple requests at once. Oshiya only needs to execute the scheduling query once to schedule hundreds of requests, thus, reducing the scheduling overhead per request and leading to a higher scalability.

1.3.3 Declarative Serializable Snapshot Isolation

Oshiya protocol implementations can be adapted easily to modified scheduling constraints. We illustrate this advantage by developing a serializable version of Snapshot Isolation called *Declarative Serializable Snapshot Isolation* (DSSI). Snapshot Isolation (SI) is a popular concurrency control protocol, but it permits non-serializable schedules that violate database integrity. Fekete et al. [FLO$^+$05] showed that every non-serializable SI schedule necessarily contains an access pattern, to which we refer to as a pivot structure. Cahill et al. [CRF09] presented the Serializable Snapshot Isolation (SSI) protocol that ensures serializable schedules by preventing such structures. We leverage the ideas of SSI, define pivot structures and show how to detect and prevent pivot structures in schedules using Oshiya. We develop the Declarative Serializable Snapshot Isolation protocol, a declarative technique that guarantees serializable schedules by preventing pivot structures while maintaining the advantages of SI. We formalize DSSI as an Oshiya protocol specification and develop an SQL implementation of DSSI using Oshiya. Our approach requires no analysis of application programs or changes to the underlying DBMS. Our implementation is concise and close to the formal protocol specification which enables us to prove that DSSI ensures serializable schedules.

1.3.4 Resource Acquisition Protocol

We introduce the *resource acquisition protocol* (RAP), a protocol not supported by DBMSs that is specifically tailored for scheduling concurrent transactions that try to acquire resources, whereby, each resource is only available in a certain quantity, like in the scenario described in Section 1.2. RAP leverages the advantages of SS2PL and SI, and provides low blocking and low abort rates. Similar to SI, customers that only browse (read) the available resources do not block other customers and are not blocked by other customers, thus, long browsing phases or inactivity periods of customers during browsing do not affect concurrency. RAP prevents deadlocks by using solely exclusive locks and by pre-ordering the resources that are acquired. We introduce the acquisition graph, a new data structure to analyze schedules of acquisition processes produced by single- and multiversion protocols. Using the acquisition graph, we prove that RAP yields fewer histories that are aborted due to insufficient resource availabilities than SI; and we show that RAP results in less blocking than SS2PL. We implement RAP using Oshiya and compare its performance to DSS2PL and an Oshiya implementation of SI. Our experiments confirm that RAP performs better than these two protocols with respect to aborts and blocking for various workloads.

1.3.5 The Oshiya Debugger and Analyzer

We present the *Oshiya Debugger and Analyzer*, a tool for debugging, visualizing, and comparing scheduling protocols developed using the Oshiya scheduling model. ODA simulates the execution of scheduling protocols over user-provided workloads. To compare protocols, ODA supports the simultaneous execution of multiple scheduling protocols over the same workload. ODA supports typical debugging features such as stepping through a protocol execution, and breakpoints. Breakpoints are modeled as break queries that stop scheduling of requests when a certain event occurs. For instance, a breakpoint can be set if a deadlock occurs. To provide contextual information, the results of a break query are visualized by highlighting matching tuples in the scheduling state. Matching tuples are identified through analyze queries. For instance, the request tuples that form the deadlock can be highlighted. The combination of break and analyze queries allows to detect and analyze errors in protocol executions and to get an understanding of how a protocol behaves for a certain workload. ODA provides the user with navigational controls to debug a protocol, e.g., after execution was stopped by a break query. These controls allow the user to browse forward and backward through protocol executions in order to (1) understand how

a detected pattern occurred and (2) to see how the protocol handles the pattern, e.g., how the protocol resolves the deadlock. ODA automatically keeps default global statistics about a protocol execution, e.g., size of scheduling relation state. Furthermore, ODA allows the user to register new measures and it permits a visual analysis of protocol execution statistics. We demonstrate the features of the system by comparing the Oshiya implementations of Snapshot Isolation and Serializable Snapshot Isolation using a banking scenario.

1.4 Organization of the Thesis

This thesis is based on an integrated collection of papers. The papers were modified to reduce overlapping sections. To avoid repetitions, the sections that describe Oshiya have been omitted in Chapter 3 and Chapter 6. The chapters adhere to the same terminology. A bibliography for all chapters is given at the end of the thesis.

Chapter 2 The Oshiya Scheduling Model

This chapter is based on results of the papers listed below.

Christian Tilgner. Declarative Scheduling in Highly Scalable Systems. In *Proceedings of the 2010 EDBT/ICDT Workshops* (Ph.D. Workshop), EDBT '10, pages 41:1–41:6, New York, ACM, 2010.

Christian Tilgner, Boris Glavic, Michael H. Böhlen, and Carl-Christian Kanne. Correctness Proof of the Declarative SS2PL Protocol Implementation. Technical Report IFI-2010.0008, University of Zurich, Department of Informatics, Zürich, Switzerland, September 2010.

Chapter 3 Declarative Serializable Snapshot Isolation

Christian Tilgner, Boris Glavic, Michael H. Böhlen, and Carl-Christian Kanne. Declarative Serializable Snapshot Isolation. In *Proceedings of the 15th East European Conference on Advances in Databases and Information Systems*, ADBIS '11, pages 170–184, Springer, 2011.

Chapter 4 Resource Acquisition Protocol

Christian Tilgner, Michael H. Böhlen, Boris Glavic, and Carl-Christian Kanne. On

Scheduling of Resource Acquisition Processes. Submitted to the 21st ACM International Conference on Information and Knowledge Management, pages 1–10, 2012.

Chapter 5 The Oshiya Debugger and Analyzer

Christian Tilgner, Michael H. Böhlen, Boris Glavic, Carl-Christian Kanne, Patrick Leibundgut, and Luis Schüller. Debugging, Visualizing, and Comparing Scheduling Protocols. Ready for submission as demonstration paper.

Chapter 6 Smile - Declarative Scheduling Middleware

Christian Tilgner, Boris Glavic, Michael H. Böhlen, and Carl-Christian Kanne. Smile: Enabling Easy and Fast Development of Domain-Specific Scheduling Protocols. In *Proceedings of the 28th British National Conference on Databases (Poster)*, BNCOD '11, pages 128–131, Springer, 2011.

CHAPTER 2

The Oshiya Scheduling Model

Abstract

Modern server systems have to efficiently schedule large amounts of concurrent client requests under various constraints such as correctness criteria and/or service-level agreements. To reduce development complexity and increase flexibility for schedulers, we investigate declarative scheduler implementations. We introduce Oshiya, a declarative scheduling model that treats pending and historical requests as data collections and uses query processing to generate request schedules. Oshiya schedules multiple requests at once, in contrast to the state of the art of imperatively implemented per-request schedulers. We illustrate Oshiya by developing the declarative strong two-phase locking protocol (DSS2PL), an Oshiya implementation of the traditional strong (rigorous) two-phase locking protocol. We formally prove the correctness of the implementation. Experiments show that the declarative approach can compete with a highly optimized database scheduler for large numbers of concurrent requests.

2.1 Introduction

Modern application servers handle large numbers of concurrently arriving requests. These have to be scheduled according to **(1)** correctness criteria like classical serializability, **(2)** service-level agreements (SLAs), and **(3)** performance goals such as increasing locality of data access by reordering requests. The state of the art is to develop schedulers imperatively for a given type of application, such as lock-based schedulers for classical database read/write requests. The advantage of this approach is that it yields fine-tuned schedulers that satisfy application-defined constraints (consistency etc.). However, imperative implementations of schedulers can be very complex and difficult to understand, in particular if the request types and correctness criteria are less well studied, for example in custom workflow systems. As a consequence, it is difficult to verify the correctness of a scheduler implementation. Adapting the scheduler to rapidly evolving market requirements, which affect the criteria (1)-(3) presented above, results in expensive and error-prone re-implementations.

We present a declarative scheduling model called *Oshiya* as a solution that allows for an easier definition and exchange of domain-specific scheduling protocols. The main ideas of Oshiya are to treat sets of requests as data collections and to employ database query processing techniques over this request data to produce request schedules in an efficient and flexible manner. Scheduling protocols are implemented as queries that select those requests that can be executed next without violating the scheduling constraints.

This approach has several advantages [Til10]. Instead of scheduling each request on its own, we apply set-at-a-time scheduling to schedule multiple requests at once, amortizing scheduling costs. Furthermore, the declarative formulation of scheduling protocols is much more concise, easier to understand and easier to modify than an imperative scheduler implementation. Our declarative specification of scheduling constraints is executable, but very close to their formal definition. This facilitates to reason about implementations, for example to prove their correctness. The contributions are the following:

- We present Oshiya, a model for declarative scheduling and show, using the strong 2PL protocol (SS2PL) as an example, how to implement traditional scheduling protocols declaratively.
- We prove the correctness of our declarative SS2PL protocol.
- We conduct experiments showing that the declarative approach can compete with a highly

optimized database scheduler for large numbers of concurrent requests.

The remainder of the chapter is organized as follows: Section 2.2 describes SS2PL. Section 2.3 introduces the Oshiya scheduling model. Section 2.4 covers the DSS2PL implementation. Section 2.6 summarizes our experimental results. We discuss related work in Section 2.7 and draw conclusions in Section 2.8.

2.2 Background: SS2PL

The SS2PL protocol is a lock-based protocol that uses two types of locks: shared read and exclusive write locks. The locks are used to block transactions from executing requests that can cause inconsistencies. Every schedule, a sequence of executed requests (read/write/abort/commit), that fulfills the conditions imposed by SS2PL on the locking behaviour of transactions is serializable and fulfills the strictness criterion [WV02]. The SS2PL protocol specifies that for each transaction the following six conditions, to which we refer to as *SS2PL conditions*, must hold [KE06]:

1. An object has to be locked before it can be read or written.
2. A transaction may not acquire a lock if already holding the lock.
3. A transaction has to respect the locks on the relevant object held by other transactions based on the lock compatibilities given in Table 2.1 [WV02]. The table is to be read as follows: a requested lock (s denotes shared lock, x denotes exclusive lock) of transaction i on an object A may be granted if it does not conflict (+) with a currently held lock on object A of transaction j ($i \neq j$). Otherwise it conflicts (-) and the requested lock may not be granted.
4. All locking operations of a transaction have to precede the first unlock operation of this transaction. This defines two phases for each transaction, a growing and a shrinking phase.
5. A transaction has to have released all its locks at its end.
6. All locks of a transaction *t*, including its exclusive write and shared read locks, are held until *t* terminates, i.e., *t* does not release its locks before it commits resp. aborts.

		Lock requested	
		$s_i(A)$	$x_i(A)$
Lock currently held	$s_j(A)$	+	-
	$x_j(A)$	-	-
	no lock	+	+

Table 2.1: SS2PL Lock Compatibilities

2.3 The Oshiya Scheduling Model

This section introduces *Oshiya*[1], a *declarative scheduling model* [Til10] to specify and implement scheduling protocols. The main ideas of Oshiya are: (1) The state of a scheduler (including the history it produces) is represented as *scheduling relations*. (2) Oshiya formalizes a protocol as a set of constraints, called *protocol specification*, that have to hold for each generated state of the relation modeling the request history. (3) The protocol specification constraints are implemented as declarative *scheduling queries*. Request scheduling is performed in so-called *scheduling iterations* by repeatedly executing the *scheduling queries* over the *scheduling relations* to determine which of the pending requests can be added to the relation that models the history without violating the protocol specification constraints. In this section, we explain the scheduling algorithm that can be parameterized using the scheduling queries.

We also discuss how to address additional issues. For example, what to do if some pending requests cannot be executed at all? And how to determine how much of the request history is actually needed to make scheduling decisions? In traditional approaches, such as locking protocols, the relevant part of the request history is encoded in special-purpose data structures such as lock tables. This is highly efficient, but inflexible. Fine-tuned algorithms on these data structures are used to detect when pending requests are not executable, e.g., in the case of deadlocks. Oshiya uses declarative queries to describe (1) the relevant subset of the request history, and (2) the conditions for requests that cannot proceed.

Overall, our approach significantly reduces the amount of code needed to implement a scheduler, but sacrifices performance compared to hand-coded schedulers in many cases. But our experiments show that the declarative approach can compete with a highly optimized database scheduler for large numbers of concurrent requests. Applications that require specialized schedulers are often under time-to-market pressure, and SLA requirements change frequently. In such

[1] Oshiya refers to the passenger arrangement staff at Japanese train stations who help to fill a train by pushing people onto the train or guiding them to free railway cars.

cases, the simplified implementation and maintenance of domain-specific scheduling protocols and SLA requirements is very useful.

2.3.1 Scheduling Relations

We model the state of a scheduler, i.e., requests to schedule as well as history information needed for scheduling decisions, as instances of three *scheduling relations*: *PendingRequests* ($\mathcal{R}$) buffers arriving client requests for scheduling. *Executable* ($\mathcal{E}$) buffers requests that have been scheduled for execution. *RelevantHistory* ($\mathcal{H}$) represents already executed requests in their execution order and models the schedule generated so far. $\mathcal{H}$ is needed because most scheduling constraints cannot be evaluated without this information. In Oshiya implementations of lock-based protocols we do not store locks explicitly. Instead, we use query processing to determine on the fly which transactions *logically* hold locks on which objects solely based on information in relation $\mathcal{H}$.

The schemata of scheduling relations $\mathcal{R}$, $\mathcal{H}$ and $\mathcal{E}$ have to be capable of representing all request information which are necessary for making scheduling decisions. The concrete schemata depend on the actual protocol resp. application domain. The basic parts of the schemata are the same for most of the scheduling protocols. A standard request consists of an identifier for the transaction of the request, the type of request (e.g., read/write) and the object the request is applied to. Furthermore, the objects in $\mathcal{H}$ and $\mathcal{E}$ have to have an ID that is used to record the execution order of requests.

Example 1. For presentation purposes, we use simplified schemata for the scheduling relations in this example. Assume the following schema for relation $\mathcal{R}$: (TA, Op, Ob). For each request, TA is the transaction executing the request, Op is the type of request (e.g., *r* for a read), and Ob is the data object the request accesses. Relations $\mathcal{H}$ and $\mathcal{E}$ have an additional attribute ID for recording the request execution order. Using this schema, the initial scheduler state is as shown in Figure 2.1.

$\mathcal{R}_0$

TA	Op	Ob

$\mathcal{H}_0$

ID	TA	Op	Ob

$\mathcal{E}_0$

ID	TA	Op	Ob

Figure 2.1: Initial Scheduler State

2.3.2 Scheduling Queries

$Q_{Schedule}$ is the core of each Oshiya protocol implementation. It is used to determine which requests can be executed next based on the pending requests in relation $\mathcal{R}$ and on the requests that have been executed during earlier scheduling iterations (relation $\mathcal{H}$). Thus, this query implements the hard constraints (e.g., serializability) and soft constraints (such as prioritizing requests of a certain type) of the applied scheduling protocol by selecting those statements from relation $\mathcal{R}$ that can safely be executed without violating the constraints imposed by the protocol. For example, for a lock-based protocol, this query selects the pending requests that are allowed to run according to the locks held by running transactions. Examples for soft constraints are SLAs like "Do not select requests from normal customers if there are pending requests of premium customers".

Depending on the scheduling protocol, some pending requests may never qualify for execution. For instance, deadlocks may occur in a lock-based protocol. $Q_{Revoked}$ is used to identify blocked resp. non-executable requests. The transactions of these requests are aborted. Note that the choice when to abort a transaction is left to the scheduler developer. For instance, if deadlock detection is too expensive, $Q_{Revoked}$ can be used to abort transactions of requests that are older than a certain threshold and, thus, are likely to participate in a deadlock situation.

In each scheduling iteration, all executed requests are inserted into relation $\mathcal{H}$. Thus, the cardinality of relation $\mathcal{H}$ grows continuously with each scheduling iteration. This increases the execution time of $Q_{Schedule}$ and limits the scalability of our approach. To avoid this effect, requests represented as tuples in relation $\mathcal{H}$ that are irrelevant for scheduling decisions have to be deleted by $Q_{Irrelevant}$. Which of these requests are not needed for scheduling decisions depends on the applied scheduling protocol.

2.3.3 Scheduling Algorithm

The state of the scheduler is advanced in iterative steps by applying a generic *scheduling algorithm* shown in Figure 2.2 that evaluates the scheduling queries over the current instances of the scheduling relations. Each scheduling iteration (while loop) schedules multiple requests at once, resulting in updated instances of the scheduling relations. This is in contrast to DBMSs that schedule requests individually. The algorithm is the same for every protocol, but it is parameterized by the protocol-specific schema of the scheduling relations and the scheduling queries.

Each scheduling iteration performs the following steps: **(1)** Requests scheduled in the previous iteration are removed from $\mathcal{R}$. **(2)** Newly arrived client requests ($\mathcal{N}$) are added to $\mathcal{R}$. **(3)** $Q_{Revoked}$ determines transactions that have to be aborted since the requests cannot be executed due to constraint violations or blocking. **(4)** $Q_{Schedule}$ implements the scheduling protocol. It selects all requests from $\mathcal{R}$ that can be executed in this iteration without violating the protocol constraints. **(5)** Requests in $\mathcal{E}$ are executed and **(6)** added to $\mathcal{H}$. **(7)** $Q_{Irrelevant}$ identifies those requests from $\mathcal{H}$ that are irrelevant for future scheduling decisions, and is used to prune $\mathcal{H}$ so that it does not grow continuously.

$\mathcal{H} = \mathcal{E} = \mathcal{R} = \emptyset$
while true **do begin**
1 $\mathcal{R} = \mathcal{R} - \mathcal{E}$;
2 $\mathcal{R} = \mathcal{R} \cup \mathcal{N}$;
3 $\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$;
4 $\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$;
5 $Execute(\mathcal{E})$;
6 $\mathcal{H} = \mathcal{H} \cup \mathcal{E}$;
7 $\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$;
end

Figure 2.2: Generic Oshiya Scheduling Algorithm

Example 2. Reconsider the scheduler state from Example 1. Assume one new read request $r_1(x)$ of transaction t_1 got inserted in relation $\mathcal{R}$ at the beginning of scheduling iteration 1 and $Q_{Schedule}$ selected this request for execution. The scheduler state after scheduling the first request is as shown in Figure 2.3. At the beginning of scheduling iteration 2, two new requests got inserted in relation $\mathcal{R}$: $r_1(y)$, $r_2(x)$. Assume that running the scheduling queries selected both request from relation $\mathcal{R}$ for execution. This leads to the updated scheduler state illustrated in Figure 2.4.

$\mathcal{R}_1$

TA	Op	Ob
1	r	x

$\mathcal{H}_1$

ID	TA	Op	Ob
1	1	r	x

$\mathcal{E}_1$

ID	TA	Op	Ob
1	1	r	x

Figure 2.3: Example Scheduler State at the End of Scheduling Iteration 1

Applying the scheduling queries to a set of newly arrived requests $\mathcal{N}$, each scheduling iteration produces new instances of the scheduling relations $\mathcal{R}$, $\mathcal{H}$ and $\mathcal{E}$. This yields a sequence of states of $\mathcal{H}$ called history, defined below. We use this definition of history to reason over the properties of a protocol and to prove the correctness of a scheduler implementation.

$\mathcal{R}_2$

TA	Op	Ob
1	r	y
2	r	x

$\mathcal{H}_2$

ID	TA	Op	Ob
1	1	r	x
2	1	r	y
3	2	r	x

$\mathcal{E}_2$

ID	TA	Op	Ob
2	1	r	y
3	2	r	x

Figure 2.4: Example Scheduler State at the End of Scheduling Iteration 2

Definition 1 (History)**.** Let $\mathbb{I} =< \mathcal{N}_0, \dots >$ be a sequence of sets of input requests and q be protocol-specific versions of the scheduling queries. We define the history generated according to q over input $\mathbb{I}$ as $\mathbb{H}_q(\mathbb{I}) =< \mathcal{H}_0, \dots >$, where $\mathcal{H}_i$ is the state of relation $\mathcal{H}$, called a history state, after the i^{th} scheduling iteration produced using q to parameterize the generic algorithm and $\mathcal{N}_i$ as input $\mathcal{N}$. We drop q and $\mathbb{I}$ if it is clear from the context and solely use $\mathbb{H}$.

In the following, we use $\mathcal{H}$ to denote both the history relation and one history state and drop indices on $\mathcal{H}$ if the scheduling iteration is irrelevant for the discussion (same holds for $\mathcal{R}$, $\mathcal{E}$ and $\mathcal{N}$). According to the algorithm presented above, the history state $\mathcal{H}_i$ is a *cumulative snapshot*, i.e., it includes all previous history states $\mathcal{H}_j$ with $j < i$ (assumed $Q_{Irrelevant}$ is not applied).

Example 3. For instance, the history states shown in Figure 2.5 could be the result of scheduling the requests $\mathbb{I} =< \{(1, r, x), (2, r, x)\}, \{(1, r, y)\}, \{(2, r, y)\} >$.

$\mathcal{H}_0$

ID	TA	Op	Ob

$\mathcal{H}_1$

ID	TA	Op	Ob
1	1	r	x

$\mathcal{H}_2$

ID	TA	Op	Ob
1	1	r	x
2	1	r	y
3	2	r	x

$\mathcal{H}_3$

ID	TA	Op	Ob
1	1	r	x
2	1	r	y
3	2	r	x
4	2	r	y

Figure 2.5: Example History States

We model a protocol as *protocol specification* defined in Definition 2. We allow quantification over scheduling iterations to enable, e.g., constraints that check the order of requests in the history. A protocol specification for DSS2PL is given in Section 2.5.

Definition 2 (Protocol Specification)**.** A protocol specification Φ is a set of protocol specification constraints formulated as first-order predicate logic expressions over $\mathbb{H}$.

The formalization of a protocol as logical constraints and its implementation as queries allows us to formally reason about the correctness of an implementation. Given a protocol specification Φ and an implementation of this protocol as a set q of scheduling queries, the definition presented

below defines what it means for q to correctly implement Φ. This is the case if for every input $\mathbb{I}$, the history created by our scheduling algorithm using q satisfies Φ.

Definition 3 (Correctness of Scheduling Queries)**.** Scheduling queries q satisfy a protocol specification Φ, denoted as $q \models \Phi$, if for every input sequence $\mathbb{I}$ the generated history $\mathbb{H}$ produced using q satisfies Φ: $\mathbb{H}_q(\mathbb{I}) \models \Phi$.

2.3.4 Assumptions and Notational Remarks

In this thesis, we make the following assumptions: (1) We limit the types of requests that have to be scheduled to atomic database requests, i.e., a read request of a transaction *i* on object *x* denoted as $r_i(x)$, a write request $w_i(x)$, an abort request a_i, and a commit request c_i. The decision to use only such requests is based on the fact that web applications typically manipulate one record at a time [CRS+08]. For example, most of the Amazon services store and retrieve data by primary key only (requests specify the primary key) and do not require complex querying and full DBMS functionality [DHJ+07, Vog07]. (2) A transaction waits until its current request is executed before issuing new requests. (3) Object identifiers are unique over all relations. (4) Rollbacks of transactions are considered as regular requests issued by clients. Extending Oshiya to schedule complex queries like joins or range queries is an interesting avenue for future work. Assumptions 2-4 simplify the presentation, but can be changed with minor modifications to Oshiya.

Scheduling queries are given as domain relational calculus (DRC) expressions. Capital letters denote variables, small letters indicate constants, and ϵ denotes *null*. All non-target variables not used in a universal quantification are implicitly existentially quantified. For instance, instead of $\{A \mid \exists B : (I(A,B) \wedge \neg \exists C : (J(C,A)))\}$ we write $\{A \mid I(A,B) \wedge \neg J(C,A)\}$. Unrestricted existentially quantified variables are displayed as an underline ("_"), disjunctive use of constants by "|". For instance, for the expression $I(A,B) \wedge (A = a \vee A = c)$ we use the shortcut $I(a|c,\_)$. We define aggregation as: $\{G, F_1(A_1), \ldots, F_n(A_n) \mid E\}$. E is a domain relational calculus expression, G is a set of attributes on which to group on (can be empty), and each F_i is an aggregate over attribute A_i. We make use of set functions union $\cup$ and difference $-$. The use of DRC instead of SQL, as required by the Smile middleware [TGBK11b] used for the experiments, results in more concise formulations of scheduling queries. For example, using the scheduling relation schemata from Example 1, the SQL query presented below selects all requests of already committed transactions from relation $\mathcal{H}$:

```
SELECT TA,Op,Ob FROM H as h1
WHERE EXISTS ( SELECT * FROM H as h2
               WHERE h1.TA=h2.TA AND
               (h2.Op='a' OR h2.Op='c'))
```

This query can easily and shorter be formulated in DRC as:

$$\{T, A, O \mid \mathcal{H}(\_, T, A, O) \wedge \mathcal{H}(\_, T, a|c, \_)\}$$

2.4 DSS2PL

In this section, we illustrate Oshiya by developing DSS2PL, an Oshiya implementation of SS2PL. We demonstrate that our approach is expressive enough to implement traditional protocols and compare the performance with highly optimized database schedulers. Furthermore, we prove the correctness of the DSS2PL implementation (Section 2.5).

2.4.1 Scheduling Relations

For DSS2PL, we use the scheduling relation schemata shown in Figure 2.6. For each incoming request, we insert a tuple into relation $\mathcal{R}$ storing an identifier for the transaction of the request (*TA*), the position of the request in the sequence of requests forming the transaction (*Seq*), the type of request (e.g., read or write, modeled by attribute *Op*), the database object the request is applied to (*Ob*), the scheduling iteration the request arrived (*SI*), and content necessary for the request execution (e.g., values of write requests). The schema of relations $\mathcal{H}$ and $\mathcal{E}$ contain an additional attribute ID that is used to record the execution order of requests.

$\mathcal{R}$	(TA, Seq, Op, Ob, SI, C)
$\mathcal{H}$	(ID, TA, Seq, Op, Ob, C)
$\mathcal{E}$	(ID, TA, Seq, Op, Ob, C)

Figure 2.6: Schemata of DSS2PL Scheduling Relations

2.4.2 Scheduling Queries

To implement the scheduling queries for DSS2PL, we have to infer the locks held by currently running transactions. We have to guarantee that after each scheduling iteration the inferred locks and the schedule represented by relation $\mathcal{H}$ fulfill the protocol constraints. The approach we take here is to develop queries that extract locking information from relation $\mathcal{H}$. These queries are applied as subqueries in $Q_{Schedule}$ whenever access to lock information is needed to decide if a pending request can be executed so that adding the selected request to relation $\mathcal{H}$ does not break the conditions imposed by SS2PL.

Inferring Lock Information. Two queries $\mathcal{X}$ (for write locks) and $\mathcal{S}$ (for read locks) with result schema {Ob, TA} are used to infer which locks are held by which transactions. Locking information is never actually stored, but extracted from relation $\mathcal{H}$ on the fly. An object x is write-locked by a transaction t, if relation $\mathcal{H}$ contains a write request of t on x but no abort or commit request of t:

$$\mathcal{X} \;=\; \{O, T \mid \mathcal{H}(\_, T, \_, w, O, \_) \wedge \neg\mathcal{H}(\_, T, \_, a|c, \_, \_)\}$$

An object x is read-locked by a transaction t, if relation $\mathcal{H}$ contains a read request of t on x but no write request of t on x and no abort or commit request of t:

$$\mathcal{S} \;=\; \{O, T \mid \mathcal{H}(\_, T, \_, r, O, \_) \wedge \neg\mathcal{H}(\_, T, \_, w, O, \_) \wedge \neg\mathcal{H}(\_, T, \_, a|c, \_, \_)\}$$

Example 4. As an example for the evaluation of queries $\mathcal{X}$ and $\mathcal{S}$, consider the instance of relation $\mathcal{H}$ shown in Figure 2.7(a) and the results of queries $\mathcal{X}$ and $\mathcal{S}$ (Figure 2.7(b) and (c)). For this instance of relation $\mathcal{H}$, transaction 15 is (logically) locking object x exclusively. Transaction 8 is not holding locks anymore, because it already finished. Transaction 14 is (logically) holding a shared lock on object z.

(a) $\mathcal{H}$

ID	TA	Seq	Op	Ob	C
1	15	1	r	x	-
2	8	1	w	y	5
3	8	2	c	-	-
4	15	2	w	x	7
5	14	1	r	z	-

(b) $\mathcal{X}$

Ob	TA
x	15

(c) $\mathcal{S}$

Ob	TA
z	14

Figure 2.7: Example Evaluation of $\mathcal{X}$ and $\mathcal{S}$

$\boldsymbol{Q_{Schedule}}$. After defining the request database schema and inferring the lock information, we can implement $Q_{Schedule}$ which selects all executable requests from relation $\mathcal{R}$, as explained in Section 2.3.2. $Q_{Schedule}$ for DSS2PL is shown in Figure 2.8. $Q_{Schedule}$ selects those requests from relation $\mathcal{R}$ which can be executed safely without violating the SS2PL constraints defined in Section 2.2. This includes all requests contained in relation $\mathcal{R}$ except for requests in the following sets: Requests on objects that are write-locked by another transaction ($OpsOnXLO$) and write requests on objects that are read-locked by another transaction ($WOpsOnSLO$). $OpsOnXLO$ returns all requests that intend to access objects which are currently write-locked by another transaction. Similarly, $WOpsOnSLO$ returns all write requests accessing objects, currently read-locked by other transactions. Aborts as well as commits may always be selected for execution.

$$
\begin{aligned}
Q_{Schedule} &= \{GenID(), T, N, A, O, C \mid \mathcal{R}(T, N, A, O, \_, C) \wedge \\
&\qquad (MinStmtPerObj(\_, T) \vee (A = a \vee A = c))\} \\
MinStmtPerObj &= \{O, Min(T) \mid LegalOps(T, \_, O)\} \\
LegalOps &= \{T, N, O \mid \mathcal{R}(T, N, \_, O, \_, \_) \wedge \neg OpsOnXLO(T, N, O) \wedge \\
&\qquad \neg WOpsOnSLO(T, N, O)\} \\
WOpsOnSLO &= \{T, N, O \mid \mathcal{R}(T, N, w, O, \_, \_) \wedge \mathcal{S}(O, T_2) \wedge T \neq T_2\} \\
OpsOnXLO &= \{T, N, O \mid \mathcal{R}(T, N, \_, O, \_, \_) \wedge \mathcal{X}(O, T_2) \wedge T \neq T_2\} \\
\mathcal{X} &= \{O, T \mid \mathcal{H}(\_, T, \_, w, O, \_) \wedge \neg \mathcal{H}(\_, T, \_, a|c, \_, \_)\} \\
\mathcal{S} &= \{O, T \mid \mathcal{H}(\_, T, \_, r, O, \_) \wedge \neg \mathcal{H}(\_, T, \_, w, O, \_) \wedge \neg \mathcal{H}(\_, T, \_, a|c, \_, \_)\}
\end{aligned}
$$

Figure 2.8: DSS2PL $Q_{Schedule}$ Query

To cover the case where multiple pending read resp. write requests want to access the same unlocked object, we select only one request per object from relation $\mathcal{R}$. This is achieved by *MinStmtPerObj*. Otherwise, the SS2PL constraints may not hold for the resulting instance of relation $\mathcal{H}$. Assume $\mathcal{R}$ contains two requests $w_1(x)$ and $w_2(x)$ that intend to write an unlocked object x. If both requests are chosen by $Q_{Schedule}$, this results in two transactions holding a write lock on the same object which is not allowed due to SS2PL constraint 3 in Section 2.2. Even though the selection of two read requests $r_3(x)$ and $r_4(x)$ on the same unlocked object x does not lead to a violation, for simplicity we decided for the more strict strategy to choose only one request per object (e.g. $r_3(x)$). $Q_{Schedule}$ can easily be extended to be able to handle multiple read requests on the same object in the same scheduling iteration. In $Q_{Schedule}$, the GenID() function generates unique increasing numerical identifiers used to establish a total request order in $\mathcal{H}$.

$\mathcal{R}$

TA	Seq	Op	Ob	SI	C
10	2	w	y	3	7
20	2	w	x	3	8

$\mathcal{H}$

ID	TA	Seq	Op	Ob	C
1	10	1	r	x	-
2	20	1	w	y	12

Figure 2.9: Deadlock Situation

$\boldsymbol{Q_{Revoked}}$. As explained in Section 2.3.2, $Q_{Revoked}$ is used to abort transactions of pending requests that may not be executed or have been pending for too long. In a lock-based protocol, requests can be blocked infinitely if they belong to transactions that are in a deadlock situation. An example deadlock situation is shown in Figure 2.9. Transactions 10 and 20 are in a deadlock situation, because they are waiting for each other to release their lock.

$$Q_{Revoked} = \{T, N, A, O, S, C \mid \mathcal{R}(T, N, A, O, S, C) \wedge S + n <= i\}$$

Figure 2.10: DSS2PL $Q_{Revoked}$ Query

$Q_{Revoked}$ resolves deadlock situations by aborting transaction that are participating in a deadlock. Either $Q_{Revoked}$ can be used to detect which transactions belong to a deadlock and to abort some of these transactions or a heuristic strategy is used to abort transactions that are likely to be in a deadlock. While the first option has the advantage to only abort deadlocked transactions, it results in a complex and expensive $Q_{Revoked}$ query (we have implemented this strategy and tested its performance). Therefore, we opted for a simple heuristic strategy: Abort all transactions if their pending requests have not been selected since n scheduling iterations. Though this strategy may abort transactions that are not part of a deadlock, the possibility for this to happen is reasonable low if n is large enough and this disadvantage is more than outweighed by the performance gain over deadlock detection. $Q_{Revoked}$ implementing this strategy is given in Figure 2.10. We use attribute *SI* of relation $\mathcal{R}$ (see Figure 2.6) that represents the scheduling iteration when a request was added to relation $\mathcal{R}$ (is set by the scheduler); i denotes the current scheduling iteration. $Q_{Revoked}$ selects all requests that have been added more then n iterations ago.

$\boldsymbol{Q_{Irrelevant}}$. As explained in Section 2.3.2, $Q_{Irrelevant}$ is used to remove requests from relation $\mathcal{H}$ if they are not relevant for future scheduling decisions. Considering the DSS2PL protocol, all requests of already finished transactions can be deleted safely without influencing $Q_{Schedule}$, because they do not hold any locks and, thus, cannot influence the outcome of $Q_{Schedule}$. The proof is given in Section 2.5 resp. [TGBK10]. The resulting $Q_{Irrelevant}$ query is illustrated in Figure 2.11.

$$Q_{Irrelevant} = \{I, T, N, A, O, C \mid \mathcal{H}(I, T, N, A, O, C) \wedge \mathcal{H}(\_, T, \_, a|c, \_, \_, \_)\}$$

Figure 2.11: DSS2PL $Q_{Irrelevant}$ Query

2.4.3 Example Using DSS2PL Scheduling Queries

An example for the execution of the DSS2PL scheduling queries is given in Figure 2.12. Assume the state of relations $\mathcal{R}$ and $\mathcal{H}$ at the end of scheduling iteration 6, as shown in Figure 2.12. Object x is write-locked by transaction 15, object y is unlocked because transaction 8 did already commit, and object z is read-locked by transaction 14. For illustration purposes, we did not delete the requests from $\mathcal{H}$ returned by $Q_{Irrelevant}$, but marked them with grey background.

At the next scheduling iteration, five new requests are inserted into relation $\mathcal{R}$ as displayed in scheduling iteration 7 of Figure 2.12. The scheduler runs $Q_{Schedule}$, shown in Figure 2.8, which selects only three requests from relation $\mathcal{R}$ for execution, including $w_{34}(y)$, a write request of transaction 34 on object B, $r_{32}(z)$, and $r_2(v)$. These requests are chosen, because they either access unlocked objects or intend to read a read-locked object. Requests $r_{27}(x)$ and $r_{21}(x)$ are not chosen in this scheduling iteration, because they intend to access a write-locked object without holding the lock. Thus, their execution has to be delayed. Afterwards, the scheduler inserts the three selected requests $w_{34}(y)$, $r_{32}(z)$ and $r_2(v)$ into relations $\mathcal{E}$ and $\mathcal{H}$, see relation $\mathcal{H}$ in scheduling iteration 7 of Figure 2.12. The requests from relation $\mathcal{E}$ are executed against the back-end DBMS, and their results are returned to the clients.

At the next scheduling iteration, the requests chosen for execution in the previous iteration ($w_{34}(y)$, $r_{32}(z)$ and $r_2(v)$) are deleted from $\mathcal{R}$ and newly arrived requests are inserted into $\mathcal{R}$ (c_{15}, commit request of transaction 15). This yields the instance of $\mathcal{R}$ displayed in scheduling iteration 8 of Figure 2.12. In this iteration $Q_{Schedule}$ selects only c_{15} for execution. Tuple c_{15} is inserted into $\mathcal{E}$ and $\mathcal{H}$ (see relation $\mathcal{H}$ in scheduling iteration 8 of Figure 2.12).

The requests chosen for execution in the previous iteration (c_{15}) are deleted from $\mathcal{R}$. We assume that in the following scheduling iteration, no new request is inserted into $\mathcal{R}$. I.e., $\mathcal{R}$ solely contains $r_{27}(x)$ and $w_{21}(x)$. Since transaction 15 that locked object x has finished in the previous iteration, $r_{27}(x)$ and $w_{21}(x)$ do not have to be delayed anymore. But it is not possible to select both requests for execution because this results in an instance of relation $\mathcal{H}$ which violates SS2PL constraint 3 in Section 2.2. $Q_{Schedule}$ prevents such situations, because it selects only the request

Scheduling Iteration 6:

$\mathcal{R}_6$

TA	Seq	Op	Ob	SI	C

$\mathcal{H}_6$

ID	TA	Seq	Op	Ob	C
1	15	1	r	x	-
2	8	1	w	y	2
3	8	2	c	-	-
4	15	2	w	x	6
5	14	1	r	z	-

Scheduling Iteration 7:

$\mathcal{R}_7$

TA	Seq	Op	Ob	SI	C
27	1	r	x	7	-
21	1	w	x	7	7
34	1	w	y	7	4
32	1	r	z	7	-
2	1	r	v	7	-

$\mathcal{H}_7$

ID	TA	Seq	Op	Ob	C
1	15	1	r	x	-
2	8	1	w	y	2
3	8	2	c	-	-
4	15	2	w	x	6
5	14	1	r	z	-
6	34	1	w	y	4
7	32	1	r	z	-
8	2	1	r	v	-

Scheduling Iteration 8:

$\mathcal{R}_8$

TA	Seq	Op	Ob	SI	C
27	1	r	x	7	-
21	1	w	x	7	7
15	3	c	-	8	-

$\mathcal{H}_8$

ID	TA	Seq	Op	Ob	C
1	15	1	r	x	-
2	8	1	w	y	2
3	8	2	c	-	-
4	15	2	w	x	6
5	14	1	r	z	-
6	34	1	w	y	4
7	32	1	r	z	-
8	2	1	r	v	-
9	15	3	c	-	-

Figure 2.12: DSS2PL Scheduling Example Illustrating Different States of $\mathcal{R}$ and $\mathcal{H}$

of the transaction with the smallest TA value for execution if there are multiple requests accessing the same object. In this case the request $w_{21}(x)$.

2.5 Correctness of the DSS2PL Scheduling Queries

In this section, we prove the correctness of the DSS2PL scheduling queries. The DSS2PL scheduling queries are correct if after each scheduling iteration, the state of relation $\mathcal{H}$ (the generated schedule) fulfills the SS2PL conditions presented in Section 2.2. First, we recursively define the state of the request database relations after scheduling iteration $i + 1$ based on their state at scheduling iteration i and their initial state $i = 0$. Afterwards, we translate the SS2PL conditions presented in Section 2.2 into DSS2PL protocol specification constraints over the re-

quest database relations. We prove by induction over the number of scheduling iterations that these protocol specification constraints hold. To ease the presentation, we omit attributes SI and C which are not constrained.

Definition 4 (Request Database State)**.** For an input sequence <$New_0, \ldots, New_i$>, $i \in N$, where each New_x is the set of requests arriving at iteration x, we call the tuple $RDB_i = (\mathcal{H}_i, \mathcal{R}_i, \mathcal{E}_i, \mathcal{N}_i)$ the request database state after scheduler iteration i. RDB_i is defined recursively as:

$$
\begin{aligned}
i \leq 0 : \mathcal{H}_i = \mathcal{E}_i = \mathcal{R}_i = \mathcal{N}_i = \emptyset \\
i > 0 : \mathcal{R}_{i+1} = \mathcal{R}_i \cup \mathcal{N}_{i+1} - \{T, N, A, O \mid \mathcal{E}_i(\_, T, N, A, O)\} \\
\mathcal{E}_{i+1} = Q_{Schedule}(\mathcal{R}_{i+1}, \mathcal{H}_i) \\
\mathcal{H}_{i+1} = \mathcal{H}_i \cup \mathcal{E}_{i+1}
\end{aligned}
$$

An example illustrating the request database state of six scheduling iterations can be found Figures 2.13 and 2.14. As defined in Definition 4, the presented relations show the instances *at the end* of the scheduling iteration. For illustration purposes, we did not delete the irrelevant requests from $\mathcal{H}$. Instead, we solely mark them with grey background.

We often refer to requests that were executed at scheduling iteration i. We use $\mathcal{H}_i^E$ as a shortcut for $\mathcal{H}_i^E = \mathcal{H}_i - \mathcal{H}_{i-1}$ resp. $\{I, T, N, A, O \mid \mathcal{H}_i(I, T, N, A, O) \wedge \neg\mathcal{H}_{i-1}(I, T, N, A, O)\}$ with $i > 0$, i.e., the requests that got included into the history exactly at scheduler run i. We define $\mathcal{H}_i^E = \emptyset$ for $i \leq 0$.

Furthermore, we define the locks $\mathcal{X}_{i+1}$ and $\mathcal{S}_{i+1}$ as observed by $Q_{Schedule}$ at iteration $i + 1$ ($Q_{Schedule}(\mathcal{R}_{i+1}, \mathcal{H}_i)$) as follows:

$$
\begin{aligned}
\mathcal{X}_{i+1} = &\{O, T \mid \mathcal{H}_i(\_, T, \_, w, O) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c, \_)\} \\
\mathcal{S}_{i+1} = &\{O, T \mid \mathcal{H}_i(\_, T, \_, r, O) \wedge \neg\mathcal{H}_i(\_, T, \_, w, O) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c, \_))\}
\end{aligned}
$$

We assume that all transactions respect the following preconditions: **(P1)** If $\mathcal{R}$ contains a request r from transaction T then no further requests of this transaction will be added until request r is processed, i.e., a transaction waits until its current request is executed before issuing new

RDB_0

$\mathcal{N}$			
TA	**Seq**	**Op**	**Ob**

$\mathcal{R}$			
TA	**Seq**	**Op**	**Ob**

$\mathcal{E}$				
ID	**TA**	**Seq**	**Op**	**Ob**

$\mathcal{H}$				
ID	**TA**	**Seq**	**Op**	**Ob**

$\mathcal{S}$	
Ob	**TA**

$\mathcal{X}$	
Ob	**TA**

RDB_1

$\mathcal{N}$			
TA	**Seq**	**Op**	**Ob**
15	1	r	x
8	1	w	y

$\mathcal{R}$			
TA	**Seq**	**Op**	**Ob**
15	1	r	x
8	1	w	y

$\mathcal{E}$				
ID	**TA**	**Seq**	**Op**	**Ob**
1	15	1	r	x
2	8	1	w	y

$\mathcal{H}$				
ID	**TA**	**Seq**	**Op**	**Ob**
1	15	1	r	x
2	8	1	w	y

$\mathcal{S}$	
Ob	**TA**

$\mathcal{X}$	
Ob	**TA**

RDB_2

$\mathcal{N}$			
TA	**Seq**	**Op**	**Ob**
8	2	c	-
15	2	w	x
14	1	r	z

$\mathcal{R}$			
TA	**Seq**	**Op**	**Ob**
8	2	c	-
15	2	w	x
14	1	r	z

$\mathcal{E}$				
ID	**TA**	**Seq**	**Op**	**Ob**
3	8	2	c	-
4	15	2	w	x
5	14	1	r	z

$\mathcal{H}$				
ID	**TA**	**Seq**	**Op**	**Ob**
1	15	1	r	x
2	8	1	w	y
3	8	2	c	-
4	15	2	w	x
5	14	1	r	z

$\mathcal{S}$	
Ob	**TA**
x	15

$\mathcal{X}$	
Ob	**TA**
y	8

RDB_3

$\mathcal{N}$			
TA	**Seq**	**Op**	**Ob**
21	1	r	x
27	1	w	x
34	1	w	y
32	1	w	z
2	1	r	v

$\mathcal{R}$			
TA	**Seq**	**Op**	**Ob**
21	1	r	x
27	1	w	x
34	1	w	y
32	1	w	z
2	1	r	v

$\mathcal{E}$				
ID	**TA**	**Seq**	**Op**	**Ob**
6	34	1	w	y
7	2	1	r	v

$\mathcal{H}$				
ID	**TA**	**Seq**	**Op**	**Ob**
1	15	1	r	x
2	8	1	w	y
3	8	2	c	-
4	15	2	w	x
5	14	1	r	z
6	34	1	w	y
7	2	1	r	v

$\mathcal{S}$	
Ob	**TA**
z	14

$\mathcal{X}$	
Ob	**TA**
x	15

RDB_4

$\mathcal{N}$			
TA	**Seq**	**Op**	**Ob**
15	3	c	-

$\mathcal{R}$			
TA	**Seq**	**Op**	**Ob**
21	1	r	x
27	1	w	x
32	1	w	z
15	3	c	-

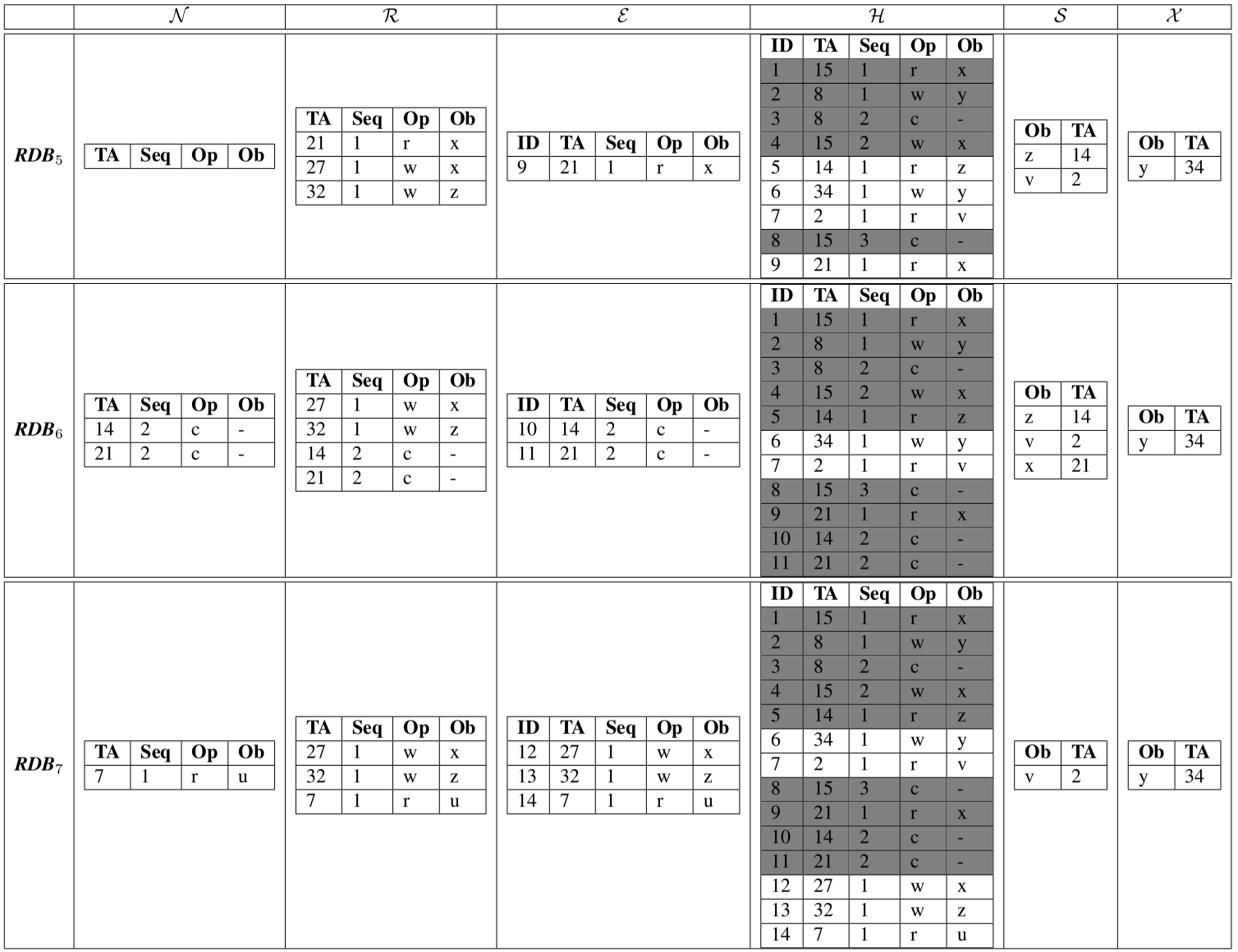
$\mathcal{E}$				
ID	**TA**	**Seq**	**Op**	**Ob**
8	15	3	c	-

$\mathcal{H}$				
ID	**TA**	**Seq**	**Op**	**Ob**
1	15	1	r	x
2	8	1	w	y
3	8	2	c	-
4	15	2	w	x
5	14	1	r	z
6	34	1	w	y
7	2	1	r	v
8	15	3	c	-

$\mathcal{S}$	
Ob	**TA**
z	14
v	2

$\mathcal{X}$	
Ob	**TA**
x	15
y	34

Figure 2.13: Example Illustrating Request Database States RDB_0-RDB_4



	$\mathcal{N}$	$\mathcal{R}$	$\mathcal{E}$	$\mathcal{H}$	$\mathcal{S}$	$\mathcal{X}$
RDB_5	**TA** **Seq** **Op** **Ob**	**TA** **Seq** **Op** **Ob** 21 1 r x 27 1 w x 32 1 w z	**ID** **TA** **Seq** **Op** **Ob** 9 21 1 r x	**ID** **TA** **Seq** **Op** **Ob** 1 15 1 r x 2 8 1 w y 3 8 2 c - 4 15 2 w x 5 14 1 r z 6 34 1 w y 7 2 1 r v 8 15 3 c - 9 21 1 r x	**Ob** **TA** z 14 v 2	**Ob** **TA** y 34
RDB_6	**TA** **Seq** **Op** **Ob** 14 2 c - 21 2 c -	**TA** **Seq** **Op** **Ob** 27 1 w x 32 1 w z 14 2 c - 21 2 c -	**ID** **TA** **Seq** **Op** **Ob** 10 14 2 c - 11 21 2 c -	**ID** **TA** **Seq** **Op** **Ob** 1 15 1 r x 2 8 1 w y 3 8 2 c - 4 15 2 w x 5 14 1 r z 6 34 1 w y 7 2 1 r v 8 15 3 c - 9 21 1 r x 10 14 2 c - 11 21 2 c -	**Ob** **TA** z 14 v 2 x 21	**Ob** **TA** y 34
RDB_7	**TA** **Seq** **Op** **Ob** 7 1 r u	**TA** **Seq** **Op** **Ob** 27 1 w x 32 1 w z 7 1 r u	**ID** **TA** **Seq** **Op** **Ob** 12 27 1 w x 13 32 1 w z 14 7 1 r u	**ID** **TA** **Seq** **Op** **Ob** 1 15 1 r x 2 8 1 w y 3 8 2 c - 4 15 2 w x 5 14 1 r z 6 34 1 w y 7 2 1 r v 8 15 3 c - 9 21 1 r x 10 14 2 c - 11 21 2 c - 12 27 1 w x 13 32 1 w z 14 7 1 r u	**Ob** **TA** v 2	**Ob** **TA** y 34

Figure 2.14: Example Illustrating Request Database States RDB_5-RDB_7

requests. **(P2)** No transaction is taking any actions after its commit or abort. These preconditions can be formulated as follows:

$$(\mathbf{P1})\ \forall T, N, i : \mathcal{R}_i(T, N, \_, \_) \Rightarrow \neg\mathcal{R}_i(T, N_2, \_, \_) \wedge N \neq N_2$$
$$(\mathbf{P2})\ \forall T, h, i : h < i \wedge \mathcal{H}_h^E(\_, T, \_, a|c, \_) \Rightarrow \neg\mathcal{H}_i^E(\_, T, \_, \_, \_)$$

Mapping the SS2PL conditions presented in Section 2.2 to a *DSS2PL protocol specification* results in the first-order predicate logic expressions C1-C6 presented in the definition below. We refer to these expressions as *DSS2PL protocol specification constraints*.

Definition 5 (DSS2PL protocol)**.** A sequence of states of relation $\mathcal{H}$ respects the DSS2PL protocol iff the following protocol specification constraints hold:

Constraint C1: An object has to be locked before it can be read or written. This condition can be expressed as follows:

$$\forall T, O, i : \mathcal{H}_i^E(\_, T, \_, r, O) \Rightarrow \mathcal{S}_{i+1}(O, T) \vee \mathcal{X}_{i+1}(O, T)$$
$$\wedge\forall T, O, i : \mathcal{H}_i^E(\_, T, \_, w, O) \Rightarrow \mathcal{X}_{i+1}(O, T)$$

We do not lock objects in advance. Instead, $Q_{Schedule}$ does not select requests accessing locked objects. Recall that locking incompatibilities between requests selected at the same scheduling iteration are prevented by *MinStmtPerObj* as explained in Section 2.4.2.

Constraint C3: A transaction has to respect the locks on the relevant object held by other transactions based on the lock compatibilities given in Table 2.1.

$$\forall O, T, T_2, i : \mathcal{X}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{i+1}(O, T_2)$$
$$\wedge\forall O, T, T_2, i : \mathcal{X}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{S}_{i+1}(O, T_2)$$
$$\wedge\forall O, T, T_2, i : \mathcal{S}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{i+1}(O, T_2)$$

Constraint C4: All locking operations of a transaction (including read and write lock operations) have to precede the first unlock operation of this transaction. This defines two phases for each

transaction, a growing and a shrinking phase. This condition is automatically fulfilled by constraint C6.

Constraint C5: A transaction has to have released all its locks at its end.

$$\forall T, h, i : h < i + 1 \wedge \mathcal{H}_h(\_, T, \_, a|c, \_) \Rightarrow \neg\mathcal{X}_{i+1}(\_, T) \wedge \neg\mathcal{S}_{i+1}(\_, T)$$

Constraint C6: All locks of a transaction are held until its end.

$$\begin{aligned}\forall T, O, g, h, i : g \leq h \leq i + 1 \wedge \mathcal{X}_g(O, T) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c) \Rightarrow \mathcal{X}_h(O, T)\\ \wedge\forall T, O, g, h, i : g \leq h \leq i + 1 \wedge \mathcal{S}_g(O, T) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c) \Rightarrow \mathcal{S}_h(O, T)\end{aligned}$$

SS2PL condition 2 is trivially fulfilled since with DSS2PL transactions do not acquire locks, $Q_{Schedule}$ determines which transactions logically hold locks, and DRC relations do not have duplicates by definition.

Furthermore, we require the lemma presented below:

Lemma 1 (One Request per Iteration and Object)**.** $Q_{Schedule}$ *guarantees that at most one request per object is scheduled during each scheduling iteration:*

$$\forall i, T, O, A : \mathcal{H}_i^E(\_, T, \_, A, O) \wedge A = r|w \wedge \mathcal{H}_i^E(\_, T_2, \_, A_2, O) \Rightarrow T = T_2 \wedge A = A_2$$

respectively

$$\forall i, T, O, A : \mathcal{H}_i^E(\_, T, \_, A, O) \wedge A = r|w \wedge (T \neq T_2 \vee A \neq A_2) \Rightarrow \neg\mathcal{H}_i^E(\_, T_2, \_, A_2, O)$$

Proof. Lemma 1 follows from the definition of *MinStmtPerObj* in the definition of $Q_{Schedule}$. We prove the claim by contradiction.

Assumption of contradiction: We assume the opposite (negation) of the condition holds for some T, O, A and i:

$$\mathcal{H}_i^E(\_, T, \_, A, O) \wedge A = r|w \wedge (T \neq T_2 \vee A \neq A_2) \wedge \mathcal{H}_i^E(\_, T_2, \_, A_2, O) \tag{2.1}$$

From the definition of $\mathcal{H}_i^E$, $\mathcal{H}_i, \mathcal{E}_i$ and $Q_{Schedule}$, we can deduce that

$$\begin{aligned}
\mathcal{H}_i^E &= \mathcal{H}_i - \mathcal{H}_{i-1} \\
&= \mathcal{H}_{i-1} \cup \mathcal{E}_i - \mathcal{H}_{i-1} \\
&= \mathcal{E}_i \\
&= Schedule(\mathcal{R}_i, \mathcal{H}_{i-1}) \\
&= \{GenId(), T, N, A, O | \mathcal{R}_i(T, N, A, O) \wedge \\
&\quad (MinStmtPerObj(_, T) \vee (A = a \vee A = c))\}
\end{aligned}$$

Which means the following equivalence holds for $\mathcal{H}_i^E$

$$\begin{aligned}
&\forall T, N, A, O : \mathcal{H}_i^E(_, T, N, A, O) \\
\Leftrightarrow\ &\mathcal{R}_i(T, N, A, O) \wedge (MinStmtPerObj(_, T) \vee (A = a \vee A = c))
\end{aligned}$$

Using this equivalence to replace $\mathcal{H}_i^E$ in Equation 2.1 we get:

$$\begin{aligned}
&\mathcal{H}_i^E(_, T, _, A, O) \wedge A = r|w \wedge (T \neq T_2 \vee A \neq A_2) \wedge \mathcal{H}_i^E(_, T_2, _, A_2, O) \\
\Leftrightarrow\ &\mathcal{R}_i(T, _, A, O) \wedge (MinStmtPerObj(_, T) \vee (A = a \vee A = c)) \\
&\wedge A = r|w \wedge (T \neq T_2 \vee A \neq A_2) \wedge \mathcal{R}_i(T_2, _, A_2, O) \wedge \\
&(MinStmtPerObj(_, T_2) \vee (A_2 = a \vee A_2 = c))
\end{aligned}$$

From $\mathcal{R}_i(T, \_, A, O) \wedge A = r|w$ follows $(A = a \vee A = c) = false$. And from $\mathcal{R}_i(T, \_, A, O) \wedge A = r|w$ and $\mathcal{R}_i(T_2, \_, A_2, O)$ we can deduce $(A_2 = a \vee A_2 = c) = false$ because $O = Null$ if $A_2 = a|c$ and this contradicts with $A = r|w$.

$$\Leftrightarrow \mathcal{R}_i(T, \_, A, O) \wedge MinStmtPerObj(\_, T) \wedge A = r|w \wedge$$
$$\mathcal{R}_i(T_2, \_, A_2, O) \wedge MinStmtPerObj(\_, T_2) \wedge (T \neq T_2 \vee A \neq A_2)$$

This implies:

$$MinStmtPerObj(O, T) \wedge MinStmtPerObj(O, T_2)$$

If $T = T_2$, then the contradiction follows from $P1$. If $T \neq T_2$, the contradiction follows from the semantics of grouping and aggregation, because aggregation produces a single result tuple for each group by value (O). □

Theorem 1 (Correctness of the DSS2PL Scheduling Queries)**.** *Each sequence of states of relation $\mathcal{H}$ <$\mathcal{H}_0$,...> generated with an arbitrary input sequence <$New_0, \ldots$> according to Definition 4 respects the DSS2PL protocol as defined by Definition 5.*

Proof. Given the recursive definition of the request database states, we prove our initial claim by induction over the number i of scheduling iterations and show that each state of relation $\mathcal{H}_i$ conforms to the DSS2PL protocol.

Therefore, we first prove that $\mathcal{H}_i$ with $i = 0$ conforms to the DSS2PL protocol. Assuming that $\mathcal{H}_i$ for scheduling iterations $i = x$ is correct, we prove the correctness of $\mathcal{H}_i$ for $i = x + 1$. Thus, all states of relation $\mathcal{H}_i$ conform to the DSS2PL protocol, i.e., protocol specification constraints C1 to C6 hold for all states of relation $\mathcal{H}_i$.

The proofs are done by transforming protocol specification constraints C1-C6 using $\mathcal{X}$, $\mathcal{S}$, P1 and P2 and by showing that C1-C6 evaluate to true for the corresponding request database state. This approach does not only apply for proving the correctness of the DSS2PL scheduling queries but also to the scheduling queries of other scheduling protocols.

Induction start

From the definition of the initial state of the request data-base we know that $\mathcal{H}_0 = \mathcal{E}_0 = \mathcal{R}_0 = \mathcal{N}_0 = \emptyset$. In the proofs for each protocol specification constraint C1 to C6, we can omit universal

quantified variables that range over the number of scheduling iterations, since for all request database relations version 0 is the only one that is defined at scheduling iteration 0.

Protocol specification constraints C1-C6 trivially evaluate to true since by definition $\mathcal{H}_0 = \mathcal{E}_0 = \mathcal{R}_0 = \mathcal{N}_0 = \mathcal{H}_0^E = \emptyset$.

Furthermore, $\mathcal{S}_i = \mathcal{X}_i = \emptyset$ for $i \leq 1$:

$$\begin{aligned}
\mathcal{X}_1 &= \{O,T|\mathcal{H}_0(_,T,_,w,O) \wedge \neg\mathcal{H}_0(_,T,_,a|c,_)\} \\
&= \{O,T|false \wedge true\} \\
&= \{O,T|false\} \\
&= \emptyset \\
\mathcal{S}_1 &= \{O,T|\mathcal{H}_0(_,T,_,r,O) \wedge \neg\mathcal{H}_0(_,T,_,w,O) \wedge \\
&\quad \neg\mathcal{H}_0(_,T,_,a|c,_))\} \\
&= \{O,T|false \wedge true \wedge true)\} \\
&= \{O,T|false)\} \\
&= \emptyset
\end{aligned}$$

Constraint C1:

$$\begin{aligned}
&\forall T,O : \mathcal{H}_0^E(_,T,_,r,O) \Rightarrow \mathcal{S}_1(O,T) \vee \mathcal{X}_1(O,T) \\
\wedge &\forall T,O : \mathcal{H}_0^E(_,T,_,w,O) \Rightarrow \mathcal{X}_1(O,T)
\end{aligned}$$

$\mathcal{H}^E$, $\mathcal{S}$, and $\mathcal{X}$ evaluate to false since $\mathcal{H}_i^E = \emptyset$ for $i \leq 0$ and $\mathcal{S}_i = \mathcal{X}_i = \emptyset$ for $i \leq 1$ by definition:

$$\begin{aligned}
&\Leftrightarrow \forall T,O : false \Rightarrow false \vee false \\
&\quad \wedge \forall T,O : false \Rightarrow false \\
&\Leftrightarrow true
\end{aligned}$$

Constraint C3:

$$\forall O, T, T_2 : \mathcal{X}_1(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_1(O, T_2)$$
$$\wedge\forall O, T, T_2 : \mathcal{X}_1(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{S}_1(O, T_2)$$
$$\wedge\forall O, T, T_2 : \mathcal{S}_1(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_1(O, T_2)$$

$\mathcal{S}$ and $\mathcal{X}$ evaluate to false since $\mathcal{S}_i = \mathcal{X}_i = \emptyset$ for $i \leq 1$ by definition:

$$\begin{aligned}
&\Leftrightarrow \forall O, T, T_2 : false \wedge T \neq T_2 \Rightarrow true \\
&\quad \wedge \forall O, T, T_2 : false \wedge T \neq T_2 \Rightarrow true \\
&\quad \wedge \forall O, T, T_2 : false \wedge T \neq T_2 \Rightarrow true \\
&\Leftrightarrow true
\end{aligned}$$

Constraint C5:

$$\forall T, h : h < 1 \wedge \mathcal{H}_h(\_, T, \_, a|c, \_) \Rightarrow \neg\mathcal{X}_1(\_, T) \wedge \neg\mathcal{S}_1(\_, T)$$

$\mathcal{H}$, $\mathcal{S}$, and $\mathcal{X}$ evaluate to false since $\mathcal{H}_i = \emptyset$ for $i \leq 0$ and $\mathcal{S}_i = \mathcal{X}_i = \emptyset$ for $i \leq 1$ by definition:

$$\begin{aligned}
&\Leftrightarrow \forall T, h : h < 0 \wedge false \Rightarrow true \wedge true \\
&\Leftrightarrow true
\end{aligned}$$

Constraint C6:

$$\forall T, O, g, h : g \leq h \leq 1 \wedge \mathcal{X}_g(O, T) \wedge \neg\mathcal{H}_0(\_, T, \_, a|c) \Rightarrow \mathcal{X}_h(O, T)$$
$$\wedge\forall T, O, g, h : g \leq h \leq 1 \wedge \mathcal{S}_g(O, T) \wedge \neg\mathcal{H}_0(\_, T, \_, a|c) \Rightarrow \mathcal{S}_h(O, T)$$

$\mathcal{H}$, $\mathcal{S}$, and $\mathcal{X}$ evaluate to false since $\mathcal{H}_i = \emptyset$ for $i \leq 0$ and $\mathcal{S}_i = \mathcal{X}_i = \emptyset$ for $i \leq 1$ by definition:

$$\begin{aligned}
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq 1 \wedge false \wedge true \Rightarrow false \\
&\quad \wedge\ \forall T, O, g, h : g \leq h \leq 1 \wedge false \wedge true \Rightarrow false \\
&\Leftrightarrow true
\end{aligned}$$

Induction step

We already proved that constraint C1 to C6 hold for relation $\mathcal{H}_i$ with $i = 0$. Assuming that constraint C1 to C6 hold for $\mathcal{H}_i$ with $i = x$, we now show that these constraints hold for $\mathcal{H}_i$ with $i = x + 1$ too. Thus, constraints C1 to C6 hold for all states of relation $\mathcal{H}$.

Constraint C1:

$$\begin{aligned}
&\forall T, O, i : \mathcal{H}_i^E(_, T, _, r, O) \Rightarrow \mathcal{S}_{i+1}(O, T) \vee \mathcal{X}_{i+1}(O, T) \\
\wedge &\forall T, O, i : \mathcal{H}_i^E(_, T, _, w, O) \Rightarrow \mathcal{X}_{i+1}(O, T)
\end{aligned}$$

We set $i = x + 1$ and remove the universal quantification on i.

$$\begin{aligned}
&\forall T, O : \mathcal{H}_{x+1}^E(_, T, _, r, O) \Rightarrow \mathcal{S}_{x+2}(O, T) \vee \mathcal{X}_{x+2}(O, T) \\
\wedge &\forall T, O : \mathcal{H}_{x+1}^E(_, T, _, w, O) \Rightarrow \mathcal{X}_{x+2}(O, T)
\end{aligned}$$

Proving first universal quantification of C1. First, we prove the first of the two universal quantifications of C1.

$$\forall T, O : \mathcal{H}_{x+1}^E(\_, T, \_, r, O) \Rightarrow \mathcal{S}_{x+2}(O, T) \vee \mathcal{X}_{x+2}(O, T)$$

Replace $\mathcal{S}$ and $\mathcal{X}$ by their definition

$$\begin{aligned}
&\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(_, T, _, r, O) \Rightarrow \\
&\quad (\mathcal{H}_{x+1}(_, T, _, r, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _)) \\
&\quad \vee (\mathcal{H}_{x+1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _))
\end{aligned}$$

To prove that this part of the constraint holds we distinguish two cases. Either $H_{x+1}(\_, T, \_, w, o)$ holds (case 1) or $\neg H_{x+1}(\_, T, \_, w, o)$ holds (case 2). Assume case 1 holds:

$$\begin{aligned}
&\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(_, T, _, r, O) \Rightarrow \\
&\quad (\mathcal{H}_{x+1}(I, T, _, r, O) \wedge false \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _) \\
&\quad \vee (true \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _)) \\
&\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(_, T, _, r, O) \Rightarrow false \vee \\
&\quad (true \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _)) \\
&\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(_, T, _, r, O) \Rightarrow \neg\mathcal{H}_{x+1}(_, T, _, a|c, _)
\end{aligned}$$

If $\neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$ holds then also $\neg(\mathcal{H}^E_u(\_, T, \_, a|c, \_) \wedge u \leq x + 1)$ holds by $\mathcal{H}_i \subseteq \mathcal{H}_{i+1}$ which follows from the recursive definition of H:

$$\begin{aligned}
&\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(_, T, _, r, O) \Rightarrow \neg\mathcal{H}_{x+1}(_, T, _, a|c, _) \wedge \\
&\quad \forall T, O, u : \neg\mathcal{H}_{x+1}(_, T, _, a|c, _) \Rightarrow \neg(\mathcal{H}^E_u(_, T, _, a|c, _) \wedge u \leq x + 1)
\end{aligned}$$

By transitivity we get:

$$\Leftrightarrow \forall T, O, u : \mathcal{H}^E_{x+1}(\_, T, \_, r, O) \Rightarrow \neg(\mathcal{H}^E_u(\_, T, \_, a|c, \_) \wedge u \leq x + 1)$$

Rewriting the implication:

$$\Leftrightarrow \forall T, O, u : \neg\mathcal{H}^E_{x+1}(\_, T, \_, r, O) \vee \neg(\mathcal{H}^E_u(\_, T, \_, a|c, \_) \wedge u \leq x + 1)$$

Applying De Morgan's law and reordering:

$$\Leftrightarrow \forall T, O, u : \neg(\mathcal{H}^E_{x+1}(\_, T, \_, r, O) \wedge u \leq x+1) \vee \neg\mathcal{H}^E_u(\_, T, \_, a|c, \_)$$

Rewriting the implication:

$$\Leftrightarrow \forall T, O, u : \mathcal{H}^E_{x+1}(\_, T, \_, r, O) \wedge u \leq x+1 \Rightarrow \neg\mathcal{H}^E_u(\_, T, \_, a|c, \_)$$

Now we have to distinguish two cases. Either $u = x+1$ holds (case 1a) or $u < x+1$ holds (case 1b). Assuming case 1a holds, both requests, $\mathcal{R}_x(\_, T, \_, r, O)$ and $\mathcal{R}_{x+1}(\_, T, \_, a|c, \_)$, would have been in relation $\mathcal{R}_{x+1}$ which contradicts precondition P1. Thus, $\neg\mathcal{H}^E_u(\_, T, \_, a|c, \_)$ and the implication evaluate to true. Assuming case 1b holds, the implication is fulfilled by definition of precondition P2.

Assume case 2 holds:

$$\begin{aligned}\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(\_, T, \_, r, O) \Rightarrow \mathcal{H}_{x+1}(I, T, \_, r, O) \wedge true \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)\\ \vee (false \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)\end{aligned}$$

Deleting irrelevant terms:

$$\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(\_, T, \_, r, O) \Rightarrow \mathcal{H}_{x+1}(I, T, \_, r, O) \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$$

A tuple of $\mathcal{H}^E_{x+1}$ is always a tuple of $\mathcal{H}_{x+1}$ by definition of $\mathcal{H}^E$ ($\mathcal{H}^E_{x+1} \subseteq \mathcal{H}_{x+1}$):

$$\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(\_, T, \_, r, O) \Rightarrow true \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$$

$\neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$ evaluates to true for the same reasons as in cases 1a and 1b:

$$\begin{aligned}&\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(I, T, \_, r, O) \Rightarrow true\\ &\Leftrightarrow true\end{aligned}$$

Proving the second universal quantification of C1. Now we prove the second quantification of constraint C1.

$$\forall T, O : \mathcal{H}^E_{x+1}(\_, T, \_, w, O) \Rightarrow \mathcal{X}_{x+2}(O, T)$$

Substitute $\mathcal{X}$ by its definition:

$$\Leftrightarrow \forall T, O : \mathcal{H}^E_{x+1}(\_, T, \_, w, O) \Rightarrow \mathcal{H}_{x+1}(\_, T, \_, w, O) \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)\}$$

If we rewrite the implication $a \rightarrow (b \wedge \neg c)$ to $(a \rightarrow b) \wedge (a \rightarrow \neg c)$ we get:

$$\begin{aligned}\Leftrightarrow \forall T, O : (\mathcal{H}^E_{x+1}(\_, T, \_, w, O) \Rightarrow \mathcal{H}_{x+1}(\_, T, \_, w, O)) \wedge (\mathcal{H}^E_{x+1}(\_, T, \_, w, O) \\ \Rightarrow \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_))\end{aligned}$$

A tuple of $\mathcal{H}^E_{x+1}$ is always a tuple of $\mathcal{H}_{x+1}$ by definition of $\mathcal{H}^E$ ($\mathcal{H}^E_{x+1} \subseteq \mathcal{H}_{x+1}$).

$$\Leftrightarrow \forall T, O : true \wedge (\mathcal{H}^E_{x+1}(\_, T, \_, w, O) \Rightarrow \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_))$$

If $\mathcal{H}^E_{x+1}(\_, T, \_, w, O)$, no $\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$ may exist for the same reasons as in cases 1a and 1b:

$$\begin{aligned}&\Leftrightarrow \forall T, O : true \wedge true \\ &\Leftrightarrow true\end{aligned}$$

Constraint C3:

$$\begin{aligned}\forall O, T, T_2, i : \mathcal{X}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{i+1}(O, T_2) \\ \wedge\forall O, T, T_2, i : \mathcal{X}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{S}_{i+1}(O, T_2) \\ \wedge\forall O, T, T_2, i : \mathcal{S}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{i+1}(O, T_2)\end{aligned}$$

We set $i = x + 1$ and prove the three universal quantified sub-expressions of C3 separately by contradiction.

$$\begin{aligned}
&\forall O, T, T_2 : \mathcal{X}_{x+2}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{x+2}(O, T_2) \\
\wedge &\forall O, T, T_2 : \mathcal{X}_{x+2}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{S}_{x+2}(O, T_2) \\
\wedge\ &\forall O, T, T_2 : \mathcal{S}_{x+2}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{x+2}(O, T_2)
\end{aligned}$$

Proving the first universal quantification of C3. We prove the first universal quantification of C3 by contradiction.

Assumption of contradiction: We assume the opposite (negation) of the first quantification holds.

$$\forall O, T, T_2 : \mathcal{X}_{x+2}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{x+2}(O, T_2)$$

Negating the expression:

$$\Leftrightarrow \mathcal{X}_{x+2}(O, T) \wedge T \neq T_2 \wedge \mathcal{X}_{x+2}(O, T_2)$$

Substituting $\mathcal{X}$ by its definition:

$$\begin{aligned}
\Leftrightarrow\ &\mathcal{H}_{x+1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _) \wedge T \neq T_2 \\
&\wedge \mathcal{H}_{x+1}(_, T_2, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T_2, _, a|c, _)
\end{aligned}$$

From Lemma 1 we know that the two requests on O have to have been scheduled during different scheduling iterations h and i, i for the request executed by T and h for the request of transaction T_2 with either $h < i$ or $i < h$. Since the use of T and T_2 in the constraint is symmetric, let wlog $h < i$. Thus, we can deduce that $i = x + 1$.

$$\begin{aligned}
\Leftrightarrow\ &h < x + 1 \wedge \mathcal{H}^E_{x+1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _) \wedge \\
&T \neq T_2 \wedge \mathcal{H}^E_h(_, T_2, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T_2, _, a|c, _)
\end{aligned}$$

From $\neg\mathcal{H}_{x+1}(\_, T_2, \_, a|c, \_)$, $h < x+1$, and the definition of $\mathcal{X}$, we follow that $\mathcal{X}_{x+1}(O, T_2)$ holds. Based on the recursive definition of $\mathcal{H}$, we know that $\mathcal{H}^E_{x+1}(\_, T, \_w, O)$ implies $LegalOps_{x+1}(T, N, O)$ which in turn implies $\neg OpsOnXLO_{x+1}(T, N, O) \wedge \neg WOpsOnSLO_{x+1}(T, N, O)$, i.e., $\mathcal{H}^E_{x+1}$ can only contain the tuple $\mathcal{H}^E_{x+1}(\_, T, \_w, O)$ if this tuple belongs to $LegalOps_{x+1}$. This stands in contradiction with $OpsOnXLO_{x+1}(T, N, O)$ which follows from $\mathcal{X}_{x+1}(O, T_2)$.

Proving second universal quantification of C3. We proceed by proving the second universal quantification of C3 by contradiction:

$$\forall O, T, T_2 : \mathcal{X}_{x+2}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{S}_{x+2}(O, T_2)$$

Substituting $\mathcal{X}$ by its definition and rewriting the implication:

$$\begin{aligned}\Leftrightarrow \forall O, T, T_2 : \neg(&\mathcal{H}_{x+1}(\_, T, \_, w, O) \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_) \wedge T \neq T_2) \\ \vee\ \neg(&\mathcal{H}_{x+1}(I, T_2, \_, r, O) \wedge \neg\mathcal{H}_{x+1}(\_, T_2, \_, w, O) \wedge \neg\mathcal{H}_{x+1}(\_, T_2, \_, a|c, \_)\end{aligned}$$

Assumption of contradiction: We assume the opposite (negation) of the quantification holds.

$$\begin{aligned}\Leftrightarrow &\mathcal{H}_{x+1}(\_, T, \_, w, O) \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c, \_) \wedge T \neq T_2 \\ &\wedge \mathcal{H}_{x+1}(I, T_2, \_, r, O) \wedge \neg\mathcal{H}_{x+1}(\_, T_2, \_, w, O) \wedge \neg\mathcal{H}_{x+1}(\_, T_2, \_, a|c, \_)\end{aligned}$$

Using the same argument as for the first quantification in C3, we can deduce that the read and write requests on O have been executed during different scheduling iterations, scheduling iteration i, for the request executed by T, and scheduling iteration h, for the request executed by $T2$, with either $h < i, i = x+1$ or $i < h, h = x+1$.

Assume case $h = x+1$ holds:

$$\begin{aligned}\Leftrightarrow &i < x+1 \wedge \mathcal{H}_i(\_, T, \_, w, O) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c, \_) \wedge T \neq T_2 \\ &\wedge \mathcal{H}_{x+1}(I, T_2, \_, r, O) \wedge \neg\mathcal{H}_{x+1}(\_, T_2, \_, w, O) \wedge \neg\mathcal{H}_{x+1}(\_, T_2, \_, a|c, \_)\end{aligned}$$

We know that $X_{x+1}(O,T)$ and the contradiction follows from

$$\begin{aligned} &\mathcal{H}_{x+1}(I,T_2,\_,r,O) \Rightarrow LegalOps_{x+1}(T_2,N,O) \\ &\Rightarrow \neg OpsOnXLO_{x+1}(T_2,N,O) \\ &\lightning \; OpsOnXLO_{x+1}(T_2,N,O) \Leftarrow X_{x+1}(O,T) \end{aligned}$$

Assume case $i = x+1$ holds:

$$\begin{aligned} \Leftrightarrow\; & h < x+1 \wedge \mathcal{H}_{x+1}(\_,T,\_,w,O) \wedge \neg\mathcal{H}_{x+1}(\_,T,\_,a|c,\_) \wedge T \neq T_2 \\ & \wedge \mathcal{H}_h(I,T_2,\_,r,O) \wedge \neg\mathcal{H}_h(\_,T_2,\_,w,O) \wedge \neg\mathcal{H}_h(\_,T_2,\_,a|c,\_) \end{aligned}$$

If $i = x+1$, we know that $\mathcal{S}_{x+1}(O,T_2)$ and the contradiction follows from:

$$\begin{aligned} &\mathcal{H}_{x+1}(I,T,\_,w,O) \Rightarrow LegalOps_{x+1}(T,N,O) \\ &\Rightarrow \neg WOpsOnSLO_{x+1}(T,N,O) \\ &\lightning \; WOpsOnSLO_{x+1}(T_2,N,O) \Leftarrow \mathcal{S}_{x+1}(O,T_2) \end{aligned}$$

Proving third universal quantification of C3. Proving the third part of C3 is redundant since the proof of the second part applies for the third part as well due to equivalence of both parts:

$$\forall O,T,T_2 : \mathcal{S}_{x+2}(O,T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{x+2}(O,T_2)$$

Rewriting the implication:

$$\Leftrightarrow \forall O,T,T_2 : \neg(\mathcal{S}_{x+2}(O,T) \wedge T \neq T_2) \vee \neg\mathcal{X}_{x+2}(O,T_2)$$

Applying De Morgan's law:

$$\Leftrightarrow \forall O, T, T_2 : \neg \mathcal{S}_{x+2}(O, T) \vee \neg (T \neq T_2) \vee \neg \mathcal{X}_{x+2}(O, T_2)$$

Rearranging the terms:

$$\Leftrightarrow \forall O, T, T_2 : \neg \mathcal{X}_{x+2}(O, T_2) \vee \neg (T \neq T_2) \vee \neg \mathcal{S}_{x+2}(O, T)$$

Applying De Morgan's law:

$$\Leftrightarrow \forall O, T, T_2 : \neg (\mathcal{X}_{x+2}(O, T_2) \wedge T \neq T_2) \vee \neg \mathcal{S}_{x+2}(O, T)$$

Rewrite as implication:

$$\Leftrightarrow \forall O, T, T_2 : \mathcal{X}_{x+2}(O, T_2) \wedge T \neq T_2 \Rightarrow \neg \mathcal{S}_{x+2}(O, T)$$

By renaming the variables in the formula, we establish the equivalence with the second part:

$$\Leftrightarrow \forall O, T, T_2 : \mathcal{X}_{x+2}(O, T) \wedge T \neq T_2 \Rightarrow \neg \mathcal{S}_{x+2}(O, T_2)$$

Constraint C5:

$$\forall T, h, i : h < i + 1 \wedge \mathcal{H}_h(\_, T, \_, a|c, \_) \Rightarrow \neg \mathcal{X}_{i+1}(\_, T) \wedge \neg \mathcal{S}_{i+1}(\_, T)$$

In constraint C5, we replace i with $x + 1$.

$$\forall T, h : h < x + 2 \wedge \mathcal{H}_h(\_, T, \_, a|c, \_) \Rightarrow \neg \mathcal{X}_{x+2}(\_, T) \wedge \neg \mathcal{S}_{x+2}(\_, T)$$

Substituting $\mathcal{X}$ and $\mathcal{S}$ by their definition:

$$\begin{aligned}
&\Leftrightarrow \forall T, h : h < x+2 \wedge \mathcal{H}_h(_, T, _, a|c, _) \Rightarrow \\
&\quad \neg(\mathcal{H}_{x+1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _)) \wedge \\
&\quad \neg(\mathcal{H}_{x+1}(I, T, _, r, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c, _))
\end{aligned}$$

Applying De Morgan's law:

$$\begin{aligned}
&\Leftrightarrow \forall T, h : h < x+2 \wedge \mathcal{H}_h(_, T, _, a|c, _) \Rightarrow \\
&\quad (\neg\mathcal{H}_{x+1}(_, T, _, w, O) \vee \mathcal{H}_{x+1}(_, T, _, a|c, _)) \wedge \\
&\quad (\neg\mathcal{H}_{x+1}(I, T, _, r, O) \vee \mathcal{H}_{x+1}(_, T, _, w, O) \vee \mathcal{H}_{x+1}(_, T, _, a|c, _))
\end{aligned}$$

If $\mathcal{H}_h(\_, T, \_, a|c, \_)$ then also $\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$ with $h < x+2$ by $\mathcal{H}_i \subseteq \mathcal{H}_{i+1}$ which follows from the recursive definition of $\mathcal{H}$.

$$\begin{aligned}
&\Leftrightarrow \forall T, h : h < x+2 \wedge \mathcal{H}_h(_, T, _, a|c, _) \Rightarrow \\
&\quad (\neg\mathcal{H}_{x+1}(_, T, _, w, O) \vee true) \wedge \\
&\quad (\neg\mathcal{H}_{x+1}(I, T, _, r, O) \vee \mathcal{H}_{x+1}(_, T, _, w, O) \vee true) \\
&\Leftrightarrow true
\end{aligned}$$

Constraint C6:

$$\begin{aligned}
&\forall T, O, g, h, i : g \leq h \leq i+1 \wedge \mathcal{X}_g(O, T) \wedge \neg\mathcal{H}_i(_, T, _, a|c) \Rightarrow \mathcal{X}_h(O, T) \\
\wedge &\forall T, O, g, h, i : g \leq h \leq i+1 \wedge \mathcal{S}_g(O, T) \wedge \neg\mathcal{H}_i(_, T, _, a|c) \Rightarrow \mathcal{S}_h(O, T)
\end{aligned}$$

Proving first universal quantification of C6. We replace the i with $x+1$ and remove the universal quantification on i.

$$\forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{X}_g(O, T) \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c) \Rightarrow \mathcal{X}_h(O, T)$$

We replace $\mathcal{X}$ by its definition:

$$\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(\_, T, \_, w, O) \wedge \neg\mathcal{H}_{g-1}(\_, T, \_, a|c, \_) \wedge \neg\mathcal{H}_{x+1}(\_, T, \_, a|c) \Rightarrow \mathcal{H}_{h-1}(\_, T, \_, w, O) \wedge \neg\mathcal{H}_{h-1}(\_, T, \_, a|c, \_)$$

From $\neg\mathcal{H}_{x+1}(\_, T, \_, a|c) \wedge g \leq h \leq x+2$ we deduce that $\neg\mathcal{H}_{h-1}(\_, T, \_, a|c, \_)$ and $\neg\mathcal{H}_{g-1}(\_, T, \_, a|c, \_)$ hold, because the highest value $g-1$ and $h-1$ can have is $x+1$.

$$\begin{aligned}
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(_, T, _, w, O) \wedge true \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c) \\
&\quad \Rightarrow \mathcal{H}_{h-1}(_, T, _, w, O) \wedge true \\
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c) \\
&\quad \Rightarrow \mathcal{H}_{h-1}(_, T, _, w, O)
\end{aligned}$$

If $\mathcal{H}_{g-1}(\_, T, \_, w, O) \wedge g \leq h \leq x+2$ holds then $\mathcal{H}_{h-1}(\_, T, \_, w, O)$ evaluates to true by $\mathcal{H}_i \subseteq \mathcal{H}_{i+1}$ which follows from the recursive definition of $\mathcal{H}$.

$$\begin{aligned}
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c) \Rightarrow true \\
&\Leftrightarrow true
\end{aligned}$$

Proving second universal quantification of C6. We replace the i with $x+1$ and remove the universal quantification on i.

$$\begin{aligned}
&\forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{S}_g(O, T) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c) \\
&\Rightarrow \mathcal{S}_h(O, T)
\end{aligned}$$

At first, we replace $\mathcal{S}$ by its definition:

$$\begin{aligned}
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(_, T, _, r, O) \wedge \\
&\quad \neg\mathcal{H}_{g-1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{g-1}(_, T, _, a|c, _) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c) \\
&\quad \Rightarrow \mathcal{H}_{h-1}(_, T, _, r, O) \wedge \neg\mathcal{H}_{h-1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{h-1}(_, T, _, a|c, _)
\end{aligned}$$

From $\neg\mathcal{H}_{x+1}(\_, T, \_, a|c) \wedge g \leq h \leq x+2$ we deduce that $\neg\mathcal{H}_{h-1}(\_, T, \_, a|c, \_)$ and $\neg\mathcal{H}_{g-1}(\_, T, \_, a|c, \_)$ hold, because the highest value $g-1$ and $h-1$ can have is $x+1$.

$$\begin{aligned}
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(_, T, _, r, O) \wedge \neg\mathcal{H}_{g-1}(_, T, _, w, O) \wedge true \wedge \\
&\quad \neg\mathcal{H}_{x+1}(_, T, _, a|c) \\
&\quad \Rightarrow \mathcal{H}_{h-1}(_, T, _, r, O) \wedge \neg\mathcal{H}_{h-1}(_, T, _, w, O) \wedge true \\
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(_, T, _, r, O) \wedge \neg\mathcal{H}_{g-1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c) \\
&\quad \Rightarrow \mathcal{H}_{h-1}(_, T, _, r, O) \wedge \neg\mathcal{H}_{h-1}(_, T, _, w, O)
\end{aligned}$$

If $\mathcal{H}_{g-1}(\_, T, \_, r, O) \wedge g \leq h \leq x+2$ holds then $\mathcal{H}_{h-1}(\_, T, \_, r, O)$ evaluates to true by $\mathcal{H}_i \subseteq \mathcal{H}_{i+1}$.

$$\begin{aligned}
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(_, T, _, r, O) \wedge \neg\mathcal{H}_{g-1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c) \\
&\quad \Rightarrow true \wedge \neg\mathcal{H}_{h-1}(_, T, _, w, O) \\
&\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(_, T, _, r, O) \wedge \neg\mathcal{H}_{g-1}(_, T, _, w, O) \wedge \neg\mathcal{H}_{x+1}(_, T, _, a|c) \\
&\quad \Rightarrow \neg\mathcal{H}_{h-1}(_, T, _, w, O)
\end{aligned}$$

Now we have to distinguish two cases. Either $\neg\mathcal{H}_{h-1}(\_, T, \_, w, O)$ holds (case 1) or $\mathcal{H}_{h-1}(\_, T, \_, w, O)$ holds (case 2).

Case 1, assuming $\neg\mathcal{H}_{h-1}(\_, T, \_, w, O)$ holds, then also $\neg\mathcal{H}_{g-1}(\_, T, \_, w, O)$ holds by $\mathcal{H}_i \subseteq \mathcal{H}_{i+1}$ which follows from the recursive definition of $\mathcal{H}$.

$$\Leftrightarrow \forall T, O, g, h : g \leq h \leq x+2 \wedge \mathcal{H}_{g-1}(\_, T, \_, r, O) \wedge true \wedge \neg \mathcal{H}_{x+1}(\_, T, \_, a|c) \Rightarrow true$$
$$\Leftrightarrow true$$

Case 2, assuming $\mathcal{H}_{h-1}(\_, T, \_, w, O)$ holds, then $\mathcal{X}_{h-1}(O, T)$ holds. And because of constraint C3, transaction T may not hold a shared lock while holding an exclusive lock. Thus, we follow that the lock hold by T changed from a shared to an exclusive lock and the first universal quantification of C6 hold, which we have proved before. □

We proved that protocol specification constraints C1-C6 hold for each state of relation $\mathcal{H}$. For simplicity, we did not yet consider the $Q_{Irrelevant}$ query, which deletes statements from relation $\mathcal{H}$ which are irrelevant for scheduling decisions. $Q_{Irrelevant}$ for DSS2PL is defined as follows:

$$Q_{Irrelevant} = \{I, T, N, A, O \mid \mathcal{H}(I, T, N, A, O) \wedge \mathcal{H}(\_, T, \_, a|c, \_)\}$$

Now, we have to show that the execution of $Q_{Irrelevant}$ has no influence on the correctness of relation $\mathcal{H}$. Therefore, we have to adapt Definition 4 to Definition 6.

Note that the step to remove revoked requests ($Q_{Revoked}$ query) from relation $\mathcal{R}$ as explained in Section 2.3.2 does not influence the correctness of $Q_{Schedule}$. This is because $Q_{Revoked}$ deletes requests which have not been scheduled and does not touch information of relation $\mathcal{H}$.

Definition 6 (Request Database State')**.** For an input sequence <$New_0, \ldots, New_i$>, $i \in N$, we call the tuple $RDB_i = (\mathcal{H}_i, \mathcal{R}_i, \mathcal{E}_i, \mathcal{N}_i)$ the request database state after scheduling iteration i. RDB_i is defined recursively as follows. Thereby, $\mathcal{T}_i$ denotes an auxiliary set to define $\mathcal{H}$.

$$\mathcal{H}_i = \mathcal{E}_i = \mathcal{R}_i = \mathcal{N}_i = \emptyset, for\, i \leq 0$$

$$\begin{aligned}
\mathcal{R}_{i+1} &= \mathcal{R}_i \cup \mathcal{N}_{i+1} - \{T, N, A, O \mid \mathcal{E}_i(_, T, N, A, O)\} \\
\mathcal{E}_{i+1} &= Q_{Schedule}(\mathcal{R}_{i+1}, \mathcal{H}_i) \\
\mathcal{T}_{i+1} &= \mathcal{H}_i \cup \mathcal{E}_{i+1} \\
\mathcal{H}_{i+1} &= \mathcal{T}_{i+1} - Q_{Irrelevant}(\mathcal{T}_{i+1})
\end{aligned}$$

We make use of the following Lemma:

Lemma 2 (Transaction Completeness of $Q_{Irrelevant}$)**.** *The $Q_{Irrelevant}$ query selects all or none of the requests of a transaction T. Let $Q_{Irrelevant\,i} = Q_{Irrelevant}(T_i)$.*

$$\begin{aligned}
\forall T, i : &Q_{Irrelevant\,i}(_, T, _, _, _) \wedge \mathcal{H}_{i-1}(I, T, N, A, O) \\
&\Rightarrow Q_{Irrelevant\,i}(I, T, N, A, O)
\end{aligned}$$

Proof. We prove the claim by contradiction. Assumption of contradiction: There exists a transaction T and scheduling iteration i such that $Q_{Irrelevant\,i}$ has selected a request from T, but at least one request from T was not selected:

$$\begin{aligned}
&Q_{Irrelevant\,i}(_, T, _, _, _) \wedge \mathcal{H}_{i-1}(I, T, N, A, O) \\
&\Rightarrow \neg Q_{Irrelevant\,i}(I, T, N, A, O)
\end{aligned}$$

Substituting the definition of $Q_{Irrelevant}$ we get:

$$\begin{aligned}
&\mathcal{H}_{i-1}(_,T,_,_,_) \wedge \mathcal{H}_{i-1}(_,T,_,a|c,_)\} \wedge \mathcal{H}_{i-1}(I,T,N,A,O) \\
&\Rightarrow \neg\mathcal{H}_{i-1}(I,T,N,A,O) \vee \neg\mathcal{H}_{i-1}(_,T,_,a|c,_) \\
\Leftrightarrow&\mathcal{H}_{i-1}(_,T,_,_,_) \wedge \mathcal{H}_{i-1}(_,T,_,a|c,_)\} \wedge \mathcal{H}_{i-1}(I,T,N,A,O) \\
&\Rightarrow \neg\mathcal{H}_{i-1}(I,T,N,A,O) \vee \neg\mathcal{H}_{i-1}(_,T,_,a|c,_)
\end{aligned}$$

The contradiction follows from the fact the both $\mathcal{H}_{i-1}(I,T,N,A,O)$ and $\neg\mathcal{H}_{i-1}(I,T,N,A,O)$ and $\mathcal{H}_{i-1}(\_,T,\_,a|c,\_)$ and $\neg\mathcal{H}_{i-1}(\_,T,\_,a|c,\_)$ exclude each other. □

Theorem 2 (Equivalence of Definition 4 and 6)**.** *Each sequence of states <$\mathcal{H}_0$,...> of relation $\mathcal{H}$ generated with an arbitrary input sequence <New_0,...> according to Definition 4 is equal to a generation according to Definition 6.*

Proof. Relation $\mathcal{H}$ gets changed by (a) the insertion of the statements selected by $Q_{Schedule}$ and (b) the deletion of statements of unfinished transactions by $Q_{Irrelevant}$. We have to show two facts: (1) $Q_{Irrelevant}$ does not influence the set of requests selected by $Q_{Schedule}$. (2) Protocol specification constraints C1-C6 hold for the states of relation $\mathcal{H}$ despite of the execution of $Q_{Irrelevant}$.

Fact 1: In $Q_{Schedule}$, only the selection of read and write requests is based on information of relation $\mathcal{H}$, more precisely $\mathcal{S}$ and $\mathcal{X}$. $\mathcal{S}$ and $\mathcal{X}$ only consider statements of non-finished transactions. And $Q_{Irrelevant}$ only deletes statements of already finished transactions:

$$Q_{Irrelevant} = \{I,T,N,A,O \mid \mathcal{H}(I,T,N,A,O) \wedge \mathcal{H}(\_,T,\_,a|c,\_)\}$$

We prove that $Q_{Irrelevant}$ executed in scheduling iteration i does not change $\mathcal{X}_{i+1}$ by contradiction.

$$\mathcal{X}_{i+1}=\{O,T \mid \mathcal{H}_i(\_,T,\_,w,O) \wedge \neg\mathcal{H}_i(\_,T,\_,a|c,\_)\}$$

I.e. the following rule holds for $\mathcal{X}$:

$$\forall O, T, i : \mathcal{X}_{i+1}(O, T) \Rightarrow \mathcal{H}_i(\_, T, \_, w, O) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c, \_)$$

Assumption of contradiction: We assume that a transaction lost a write lock after $Q_{Irrelevant}$ has been executed.

Since by definition $Q_{Irrelevant}$ only deletes tuples, it has to have deleted a write statement. But also by definition, $Q_{Irrelevant}$ only deletes statements if $\mathcal{H}_i(\_, T, \_, a|c, \_)$ holds. Thus, the contradiction follows from:

$$\neg\mathcal{H}_i(\_, T, \_, a|c, \_) \lightning \mathcal{H}_i(\_, T, \_, a|c, \_)\}$$

We prove that $Q_{Irrelevant}$ executed in scheduling iteration i does not change $\mathcal{S}_{i+1}$ by contradiction.

$$\mathcal{S}_{i+1}=\{O, T \mid \mathcal{H}_i(\_, T, \_, r, O) \wedge \neg\mathcal{H}_i(\_, T, \_, w, O) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c, \_)\}$$

I.e. the following rule holds for $\mathcal{X}$:

$$\forall O, T, i : \mathcal{S}_{i+1}(O, T) \Rightarrow \mathcal{H}_i(\_, T, \_, r, O) \wedge \neg\mathcal{H}_i(\_, T, \_, w, O) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c, \_)$$

Assumption of contradiction: We assume that a transaction lost a read lock after $Q_{Irrelevant}$ has been executed.

Since by definition $Q_{Irrelevant}$ only deletes tuples, it has to have deleted a read statement. But since by definition, $Q_{Irrelevant}$ only deletes statements if $\mathcal{H}_i(\_, T, \_, a|c, \_)$ holds. Thus, the contradiction follows from:

$$\neg\mathcal{H}_i(\_, T, \_, a|c, \_) \lightning \mathcal{H}_i(\_, T, \_, a|c, \_)$$

From this we follow that Fact 1 holds.

Fact 2: Now we have to show that $Q_{Irrelevant}$ does not violate any proof of constraints C1-C6.

Constraint C1:

$$\begin{aligned}&\forall T, O, i : \mathcal{H}_i^E(\_, T, \_, r, O) \Rightarrow \mathcal{S}_{i+1}(O, T) \vee \mathcal{X}_{i+1}(O, T)\\ \wedge &\forall T, O, i : \mathcal{H}_i^E(\_, T, \_, w, O) \Rightarrow \mathcal{X}_{i+1}(O, T)\end{aligned}$$

Assume constraint C1 holds for some i after Definition 4. We have to show that after deleting the requests from $Q_{Irrelevant}$ this constraint still holds. If $Q_{Irrelevant}$ deletes any request of a transaction T from the history, it deletes all requests of T. Therefore, if $Q_{Irrelevant}$ deletes all reads of transaction T on object O (changing the evaluation of the left hand side of the implication to false: $\mathcal{H}_i(\_, T, \_, r, O)$), then also the right hand side of the implication evaluates to false, because $\mathcal{S}_{i+1}(O, T)$ requires $\mathcal{H}_i(\_, T, \_, r, O)$ or $\mathcal{H}_i(\_, T, \_, w, O)$ and $\mathcal{X}_{i+1}(O, T)$ requires $\mathcal{H}_i(\_, T, \_, w, O)$. If $Q_{Irrelevant}$ deletes writes of transaction T on object O (changing the evaluation of the left hand side of the implication of the second universal quantification to false: $\mathcal{H}_i(\_, T, \_, w, O)$), then also the right hand side of the implication evaluates to false, because $\mathcal{X}_{i+1}(O, T)$ requires $\mathcal{H}_i(\_, T, \_, w, O)$. Controversially, if the deletions of $Q_{Irrelevant}$ cause the right hand side to evaluate to true, then we can use the same argument to show that the left hand side does too.

Constraint C3:

$$\begin{aligned}&\forall O, T, T_2, i : \mathcal{X}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{i+1}(O, T_2)\\ \wedge &\forall O, T, T_2, i : \mathcal{X}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{S}_{i+1}(O, T_2)\\ \wedge &\forall O, T, T_2, i : \mathcal{S}_{i+1}(O, T) \wedge T \neq T_2 \Rightarrow \neg\mathcal{X}_{i+1}(O, T_2)\end{aligned}$$

Assume constraint C3 holds for some i after Definition 4. We have to show that after deleting the requests from $Q_{Irrelevant}$ this constraint still holds. Assume $Q_{Irrelevant}$ deletes all or non requests a transaction T from the history, the right side of the implications do not change. The lefts sides of the implications can only change to false. Hence, the case $true \Rightarrow false$ is not possible.

Constraint C5:

$$\forall T, h, i : h < i + 1 \wedge \mathcal{H}_h(\_, T, \_, a|c, \_) \Rightarrow \neg\mathcal{X}_{i+1}(\_, T) \wedge \neg\mathcal{S}_{i+1}(\_, T)$$

Assume constraint C5 holds for some i after Definition 4. We have to show this constraint still holds after Definition 6. If $Q_{Irrelevant}$ deletes any request of a transaction T from the history, it deletes all requests of T. Therefore, if $Q_{Irrelevant}$ deletes an abort or commit statement of transaction T (changing the evaluation of the left hand side of the implication to false), then the right hand side of the implication does not change. This is because $Q_{Irrelevant}$ solely deletes requests and due to P3. Thus, C5 evaluates to true.

In the induction step for constraint C5, we stated that if $\mathcal{H}_h(\_, T, \_, a|c, \_) \wedge h < x + 1$ holds then also $\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$ holds, as shown in the following excerpt of the proof of constraint C5:

$$\begin{aligned}
&\Leftrightarrow \forall T, h : h < x + 2 \wedge \mathcal{H}_h(_, T, _, a|c, _) \Rightarrow \\
&\quad (\neg\mathcal{H}_{x+1}(_, T, _, w, O) \vee \mathcal{H}_{x+1}(_, T, _, a|c, _)) \wedge \\
&\quad (\neg\mathcal{H}_{x+1}(I, T, _, r, O) \vee \mathcal{H}_{x+1}(_, T, _, w, O) \vee \mathcal{H}_{x+1}(_, T, _, a|c, _))
\end{aligned}$$

If $\mathcal{H}_h(\_, T, \_, a|c, \_)$ then also $\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$ with $h < x + 2$ by $\mathcal{H}_i \subseteq \mathcal{H}_{i+1}$ which follows from the recursive definition of $\mathcal{H}$.

$$\begin{aligned}
&\Leftrightarrow \forall T, h : h < x + 2 \wedge \mathcal{H}_h(_, T, _, a|c, _) \Rightarrow \\
&\quad (\neg\mathcal{H}_{x+1}(_, T, _, w, O) \vee true) \wedge \\
&\quad (\neg\mathcal{H}_{x+1}(I, T, _, r, O) \vee \mathcal{H}_{x+1}(_, T, _, w, O) \vee true) \\
&\Leftrightarrow true
\end{aligned}$$

But even if $Q_{Irrelevant}$ deletes the abort resp. commit statement and $\mathcal{H}_{x+1}(\_, T, \_, a|c, \_)$ evaluates to false, this does not influence the result of the proof because $Q_{Irrelevant}$ always deletes all tuples of a finished transaction by definition. Thus, the proof evaluates to true as follows:

$$
\begin{aligned}
&\Leftrightarrow \forall T, h : h < x+2 \wedge \mathcal{H}_h(\_, T, \_, a|c, \_) \Rightarrow \\
&\quad (\neg\mathcal{H}_{x+1}(\_, T, \_, w, O) \vee \mathcal{H}_{x+1}(\_, T, \_, a|c, \_)) \wedge \\
&\quad (\neg\mathcal{H}_{x+1}(I, T, \_, r, O) \vee \mathcal{H}_{x+1}(\_, T, \_, w, O) \vee \mathcal{H}_{x+1}(\_, T, \_, a|c, \_)) \\
&\Leftrightarrow \forall T, h : h < x+1 \wedge \mathcal{H}_h(\_, T, \_, a|c, \_) \Rightarrow \\
&\quad (true \vee false) \wedge \\
&\quad (true \vee false \vee false) \\
&\Leftrightarrow true
\end{aligned}
$$

Constraint C6:

$$
\begin{aligned}
\forall T, O, g, h, i : g \leq h \leq i+1 \wedge \mathcal{X}_g(O, T) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c) \Rightarrow \mathcal{X}_h(O, T) \\
\wedge \forall T, O, g, h, i : g \leq h \leq i+1 \wedge \mathcal{S}_g(O, T) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c) \Rightarrow \mathcal{S}_h(O, T)
\end{aligned}
$$

Assume constraint C6 holds for some input sequence after Definition 4. We have to show that C6 holds too after Definition 6. We prove this fact by contradiction.

Assumption of contradiction: Constraint C6 breaks for an i. I.e., the opposite holds.

$$
\begin{aligned}
(g \leq h \leq i+1 \wedge \mathcal{X}_g(O, T) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c) \wedge \neg\mathcal{X}_h(O, T)) \\
\vee (g \leq h \leq i+1 \wedge \mathcal{S}_g(O, T) \wedge \neg\mathcal{H}_i(\_, T, \_, a|c) \wedge \neg\mathcal{S}_h(O, T))
\end{aligned}
$$

We have to distinguish two cases. Either the first part evaluates to true or the second one.

First part: The first part evaluates to true if $\mathcal{X}_g(O, T)$ holds for some O and T and $\neg\mathcal{H}_i(\_, T, \_, a|c)$ and $\neg\mathcal{X}_h(O, T)$ hold as well. This is not possible because, firstly, this contradicts to C6 and, secondly, $Q_{Irrelevant}$ always deletes all tuples of T by definition, i.e., $\mathcal{X}_g(O, T)$ would not hold after an execution of $Q_{Irrelevant}$.

The second part is analog.

Thus, Fact 2 holds and $Q_{Irrelevant}$ does not influence the correctness of relation $\mathcal{H}$ and $Q_{Schedule}$ always selects the same set of statements independent of the execution of $Q_{Irrelevant}$. □

2.6 Evaluation

This section presents experimental results. For the experiments we used Smile, a database-independent middleware implementation of the Oshiya scheduling model [TGBK11b]. We conduct experiments to (1) compare the native database scheduler overhead with the declarative scheduler overhead and to (2) find out the overhead of introducing a middleware layer.

2.6.1 Experimental Setup

Hardware. We ran the experiments on two machines with a 2.8GHz single-core CPU and 2GB memory each. On the first machine, a client application was executed which simulated users each executing transactions continuously. On the second machine, we ran Smile and a commercial DBMS.

Workload. Our workload simulated a bonus payment transaction over a single relation with user accounts and their balance. We ran several experiments and varied the accounts t to update per transaction and the database cardinality c. Each transaction consisted of t SELECT and t UPDATE statements executed against a single table with cardinality c. Each statement pair affected exactly one random row, with a uniform probability for each row. We used a database instance that fitted in the database buffer.

2.6.2 Method

We executed experiments in three different modes: The simulated clients connect directly to the DBMS and execute their transactions under isolation level serializable, denoted as *direct-connect mode (DC)*. The simulated clients connect to Smile. Smile maintains one database connection for each connected client and solely forwards the requests to the DBMS for execution without performing scheduling. The DBMS is taking care of scheduling the requests coming from Smile. This mode is denoted as *multi-user mode (MU)*. MU applies isolation level serializable. In the *declarative scheduling mode (DS)*, the simulated clients connect to Smile. Smile applies declarative scheduling using the DSS2PL scheduling queries presented in Section 2.4.2 which corresponds to isolation level serializable. By analyzing MU and DS, we are able to compare the performance of the native database scheduler with the declarative scheduler. Comparing DC with MU allows us to study the overhead introduced by the Smile middleware layer.

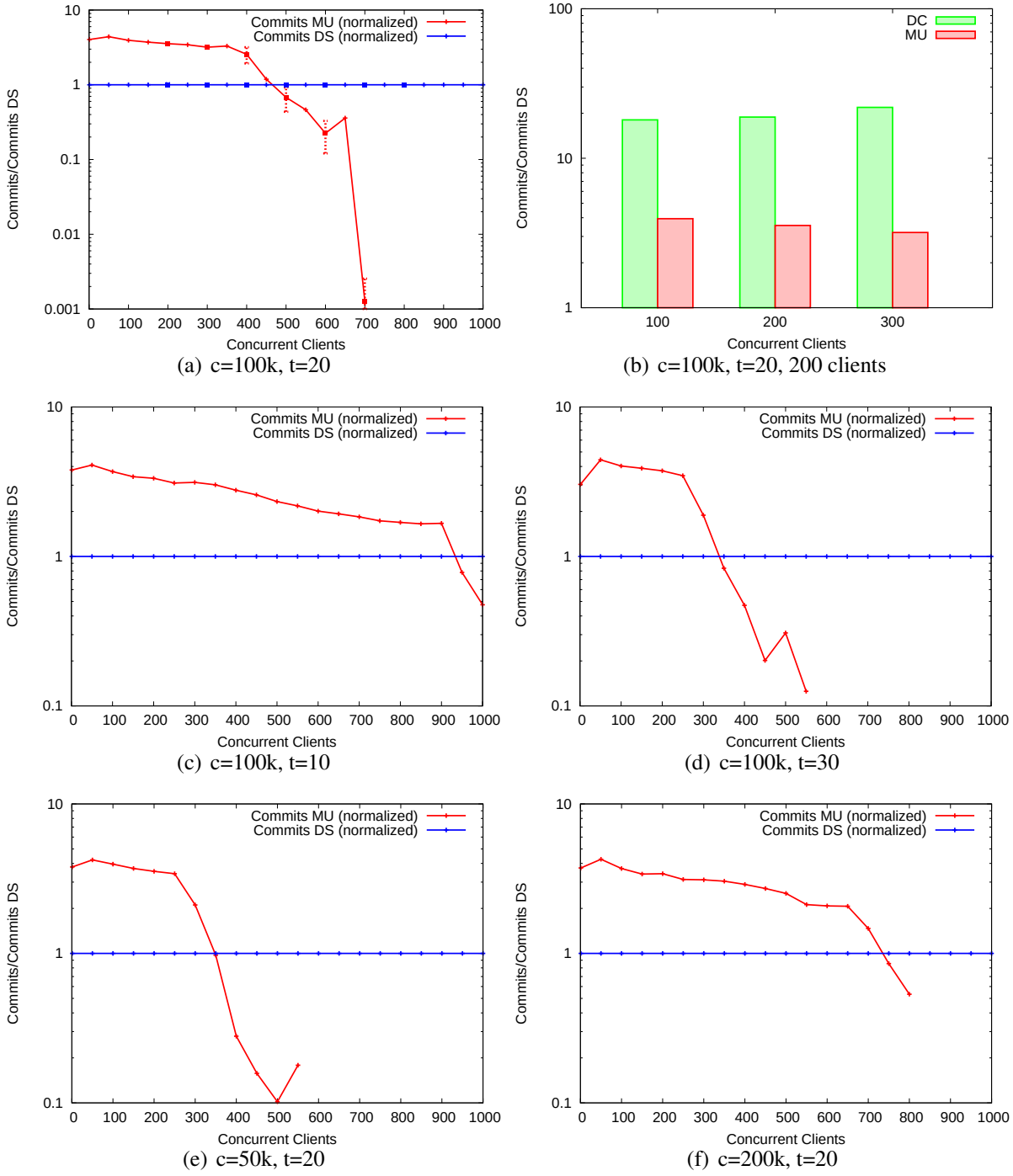
After an adequate warm-up time (determined through extensive experiments), we measured for each mode how many transactions were committed within 240s by a certain number of concurrently active clients. This method is equivalent to measuring the execution time of a fixed number of transactions but has the advantage of constant experiment durations. We repeated each individual experiment multiple times and averaged the results.

2.6.3 Results

To analyze in which client ranges our declarative scheduling approach works well, we measured the throughput of MU and DS for 1 to 1000 concurrent clients (step size 50) and ran experiments for different database cardinalities c and transaction sizes t.

Figure 2.15(a) presents the results of an experiment comparing MU and DS with database cardinality $c = 100'000$ and $t = 20$ accounts updated per transaction. The y-axis shows the commit ratio of MU to DS in logarithmic scale, with DS normalized to 1. The x-axis denotes the number of concurrent clients. We observe that in the range of 1 to approx. 450 clients, the throughput of the native database scheduler in MU is higher than the throughput of DS. DS is less efficient in that lower client range since the applied set-at-a-time scheduling is more expensive than request-by-request scheduling applied by the standard DBMS. I.e., executing the queries $Q_{Schedule}$, $Q_{Revoked}$ and $Q_{Irrelevant}$ is quite costly for only a few concurrent requests.

Thus, our main interest regards the range with large numbers of concurrent requests. From approx. 450 concurrent clients on, our declarative scheduler approach is performing even better than the native database scheduler in MU. I.e., there are parameter ranges, where declarative scheduling performs better than the native database scheduler. This shows the disadvantage of the request-by-request scheduling approach for large amounts of concurrent requests which is one of the reasons for the limited scalability of standard DBMSs. For large amounts of concurrent requests, the scheduling overhead, lock management and conflict possibility increases enormously and also lock contention thrashing appears [WV02]. Thus, the throughput drops for MU. Whereas, for larger numbers of concurrent requests, we can see the advantage of the set-at-a-time scheduling applied by our declarative approach. We only need to execute $Q_{Schedule}$ once to schedule hundreds of requests and we do not use explicit locking which implies reduced scheduling overhead per request and which leads to a higher scalability. Thus, for DS the throughput only decreases slightly with increasing the number of clients but does not drop until 1000 clients.



(a) c=100k, t=20

(b) c=100k, t=20, 200 clients

(c) c=100k, t=10

(d) c=100k, t=30

(e) c=50k, t=20

(f) c=200k, t=20

Figure 2.15: Experimental Results

To investigate in which parameter ranges our approach works well, we varied the database cardinality from 100'000 to 50'000 and 200'000 and we changed the transaction size from 20 modified rows to 10 and 30. The results are shown in Figures 2.15(c)-2.15(f). The y-axis shows the commit ratio of MU to DS in logarithmic scale, with DS normalized to 1. The x-axis denotes the number of concurrent clients. These experiments demonstrate that MU and DS behave as expected. If we increase the database cardinality or decrease the transaction size, in MU mode, the database system scales better before the drop of the throughput occurs, because of the decreasing possibility of conflicts. On the other hand, the drop of the throughput occurs earlier (for smaller client counts) if we decrease c or increase t which increases the possibility of conflicts. Note that in all experiments we experience that the performance of MU degrades rapidly after a critical number of concurrent clients is reached whereas for DS we only observe a linear decrease in the number of clients. Even for small number of clients MU is not performing better than factor 4.5.

Figure 2.15(a) illustrates another advantage of Smile compared to the native database scheduler: The declarative scheduling approach provides a high stability and predictability of the throughput. In Figure 2.15(a), the vertical dotted lines at 200-800 (step size 100) concurrent requests show the average throughput plus or minus the standard deviation for MU and DS. Based on this we can see that the standard deviation of DS is constantly low from 1-1000 concurrent requests which allows for precise predictability, whereas, we observed a large increase of the standard deviation for increasing numbers of concurrent requests for DC and MU (for readability the DC results are not presented in the graph).

To study the middleware overhead of Smile over native database scheduling, we compared the throughput of DC, MU, and DS. Figure 2.15(b) show the respective throughputs for 100, 200 and 300 concurrent clients, with DS normalized to 1 and $c = 100'000$ and $t = 20$. By comparing DC with DS, we can deduce the overhead of Smile. We can see from Figure 2.15(a) that in this range, the native database scheduler (DC) is not performing better than factor 22 compared to our declarative scheduler (DS). We observe that the declarative scheduling overhead (DS) is not more expensive than factor 4.5 compared to the native database scheduling overhead (MU).

2.7 Related Work

The ACTA framework is an approach to formalize properties of transaction models using first-order formula over schedules [CR90]. The conciseness and clarity of the ACTA approach in-

spired us to find a way to actually implement schedulers based on declarative protocol specifications.

Another declarative approach is the BOOM project which employs the declarative language Overlog to build distributed systems [ACC+10]. It uses a declarative task scheduler allowing for an easy definition of new scheduling policies. But it focuses on scheduling MapReduce tasks instead of database requests and on scheduling policies like First-Come-First-Served instead of database consistency and/or quality of service (QoS). As a result of their study, they confirm that declarative languages are a good fit for implementing scheduling policies and facilitate an easy modification of such policies.

A lot of imperative approaches exist focusing on improving user scalability and performance of DBMSs as well as adding support for sophisticated scheduling constraints such as QoS on top of standard DBMSs: With an *external queue management system*, Schroeder et al. intend to ensure QoS targets by scheduling requests which are enqueued in an external queue, using external prioritization and adjusting the multiprogramming level of the underlying DBMS [Sch06b]. The purposes of the middleware *Ganymed* comprise load balancing, user scalability (without sacrificing consistency) and performance improvement for read requests [PA04]. The *workload management system* presented by Krompass et al. classifies queries according to their expected execution time to be able to fulfill QoS as well as recognizing and automatic handling of problem queries for online transaction processing (OLTP) and business intelligence (BI) workloads [KKDK07]. *Clustered JDBC*, a Java middleware framework for database clustering offering single database views to client applications, provides high availability and performance scalability [CMZ04]. The *gatekeeper proxy* from Elnikety et al. provides external admission control and request scheduling to improve performance [ENTZ04]. Bhatti presents a system for supporting server QoS using admission control and scheduling based on several scheduling policies to improve performance and to support distinct service levels [BF99]. *QShuffler* is a query scheduler that focuses on the problem of scheduling large batches of queries in BI settings (e.g., report generation) to minimize the total completion time [AABM08]. Database vendors have developed tools to enable QoS and to improve performance of their DBMSs, e.g., IBM DB2 Query Patroller [DB2].

Our work differs from these approaches. None of these approaches provides an easy exchange of applied scheduling protocols or the definition of new application-specific protocols by using a declarative protocol description language. Instead, these approaches use fixed, imperatively implemented algorithms and are, thus, not flexible enough to react to changing requirements

resp. business processes and cannot be adjusted to certain application-specific consistency requirements. Oshiya models the state of a scheduler using relations. The scheduling state can be represented differently by other data models supporting sets and providing a declarative query language, e.g., XML and XQuery, but this is orthogonal to our approach.

2.8 Conclusions and Future Work

In this chapter, we present the Oshiya scheduling model, our approach for declarative scheduling. We demonstrate how to implement the SS2PL protocol declaratively and prove the correctness of this implementation. The experimental evaluation demonstrates that the declarative approach can compete with a highly optimized database scheduler for large numbers of concurrent requests.

In future work, we will continue with experimental evaluations of application-specific consistency and multi-versioning protocols. The design and implementation of a declarative scheduler specification language to allow more succinct definitions of scheduling protocols and more specialized protocols supporting (e.g., SLAs) is an interesting avenue for future work.

CHAPTER 3

Declarative Serializable Snapshot Isolation

Abstract

Snapshot isolation (SI) is a popular concurrency control protocol, but it permits non-serializable schedules that violate database integrity. The Serializable Snapshot Isolation (SSI) protocol ensures (view) serializability by preventing pivot structures in SI schedules. In this work, we leverage the SSI approach and develop the Declarative Serializable Snapshot Isolation (DSSI) protocol, an SI protocol that guarantees serializable schedules. Our approach requires no analysis of application programs or changes to the underlying DBMS. We present an implementation and prove that it ensures serializability.

3.1 Introduction

Snapshot Isolation (SI) [BBG+95] is a popular multiversion concurrency control (MVCC) protocol, but it permits non-serializable schedules. Fekete et al. [FLO+05] showed that every non-serializable SI schedule necessarily contains an access pattern with two consecutive vulnerable

edges (see Section 3.2.2), and Cahill et al. [CRF09] presented the *Serializable Snapshot Isolation (SSI)* protocol that ensures serializable schedules by preventing such structures.

We leverage the ideas of SSI, define *pivot structures* and propose the *Declarative Serializable Snapshot Isolation (DSSI)* protocol, a declarative technique that guarantees serializable schedules by preventing pivot structures while maintaining the advantages of SI. We implement DSSI using our declarative scheduling model called *Oshiya* introduced in Chapter 2. Oshiya models the scheduler state (including the generated schedule) in so-called *scheduling relations* and formalizes a protocol as a *protocol specification*. A protocol specification is a set of constraints specified as first-order predicate logic expressions that have to hold for all scheduling relation states. In Oshiya, a protocol specification is implemented as declarative *scheduling queries*. Request scheduling is performed by applying a generic scheduling algorithm that repeatedly executes the scheduling queries over the scheduling relations. The queries determine the pending requests that can be added to the relation modeling the schedule without violating the protocol specification. We show how to detect and prevent pivot structures using Oshiya and implement the DSSI protocol specification as scheduling queries. Our implementation is concise and close to the formal protocol specification which enables us to prove its correctness. The main contributions are the following:

- We introduce DSSI, a protocol that ensures serializable SI executions, and formalize it as an Oshiya protocol specification.
- Using Oshiya we develop an SQL implementation of DSSI.
- We prove that the implementation ensures serializable schedules.

The chapter structure is as follows: Section 3.2 describes SI and reviews the approach applied by the SSI protocol to detect non-serializable schedules. Section 3.3 shows how we model data snapshots and presents schemata for the scheduling relations. Section 3.4 formalizes the DSSI protocol. Section 3.5 presents the DSSI scheduler implementation. Section 3.6 proves that our implementation ensures serializable executions. We review related work in Section 3.7 and conclude in Section 3.8.

3.2 Background: Snapshot Isolation and Serializability

We model a transaction t_i as a sequence of read and write requests (denoted as $r_i(x)$ resp. $w_i(x)$ where x stands for the accessed data item). Each transaction finishes with an abort (a_i) or commit (c_i) request. The write-set WS_i of t_i contains all data items written by t_i. A *history* (schedule) is a sequence of interleaved executions of requests from a set of concurrent transactions. The requests in a history are totally ordered. We write $p <^H q$ if request p is executed before request q. Let bot_i denote the begin of t_i (when t_i executed its first request) and eot_i its end (when t_i aborted resp. committed). The execution interval of a committed transaction t_i is $[bot_i, c_i]$, the one of a non-aborted, possibly committed transaction t_i is $[bot_i, l_i]$ (l_i is t_i's latest request). Two committed transactions t_i and t_j *overlapped* if: $Overlapped_{ij} \Leftrightarrow [bot_i, c_i] \cap [bot_j, c_j] \neq \emptyset$. Two non-aborted (maybe active) transactions t_i and t_j *overlap* if: $Overlap_{ij} \Leftrightarrow [bot_i, l_i] \cap [bot_j, l_j] \neq \emptyset$.

3.2.1 Snapshot Isolation

SI is a multiversion concurrency protocol that maintains multiple versions of data items (tuples). Each write $w_i(x)$ creates a new version of item x that is visible to other transactions after c_i. Each read $r_i(x)$ accesses the latest version of x written by transactions that committed before bot_i. Moreover, a transaction always sees the versions it created itself. Under SI, reads are never delayed because of write requests of concurrent transactions and vice versa. SI avoids inconsistent read anomalies because transactions never access partial results of other concurrent transactions. SI requires disjoint write-sets of concurrent committed transactions which is, e.g., ensured by the First-Committer-Wins (FCW) rule. FCW specifies that a transaction is aborted if a concurrent transaction with an overlapping write-set already committed. FCW also prevents lost updates. A typical anomaly that leads to non-serializable SI histories is the *Write Skew* [BBG+95], detailed in Example 5.

Example 5. Consider history H_{ws} in Figure 3.1. Initially, data items $x = 50$ and $y = 50$ are consistent and satisfy constraint $C = x + y \geq 0$. Transaction t_1 reads x and y. A concurrent transaction t_2 reads x and y, writes x (after subtracting 90) and commits (after checking C). Finally, t_1 writes y (after subtracting 90) and commits (after checking C). In the final state, C is violated although t_1 and t_2 checked C explicitly before committing. This happens because C is checked on the version of x and y that is visible to t_1 and t_2 and not on the final state resulting from their interleaved execution.

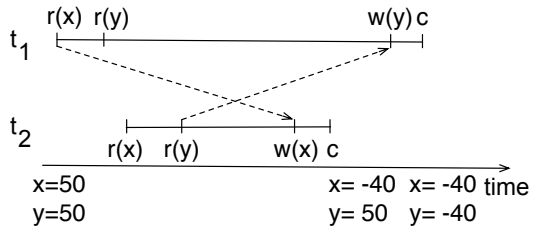


Figure 3.1: History H_{ws}

3.2.2 Detecting Non-Serializable Histories

Serializability of SI histories can be checked using a multiversion serialization graph $MVSG = (N, E)$ [CRF09]. The MVSG of a history H is a graph that contains a node for each committed transaction t_i of H: $t_i \in N \Leftrightarrow c_i \in H$. It contains an edge from committed transaction t_i to committed transaction t_j with $i \neq j$ if (a) t_i writes a version of x and t_j writes a later version of x, (b) t_i writes a version of x and t_j reads this or a later version of x, and (c) t_i reads a version of x and t_j writes a later version of x. An edge of type (c) that occurs between two overlapped transactions t_i and t_j is called a *vulnerable edge* [FLO$^+$05]. A *pivot structure* exists between three committed transactions t_i, t_j and t_k (t_i and t_k are not necessarily distinct) if there is a vulnerable edge between t_i and t_j and between t_j and t_k. Fekete et al. [FLO$^+$05] showed that every MVSG of a non-serializable SI history must contain a pivot structure. The existence of a pivot structure is a necessary but not sufficient condition for the non-serializability of an SI history. Thus, an SI history is serializable if its MVSG does not contain pivot structures.

Example 6. Figure 3.1 shows history H_{ws}. Vulnerable edges are shown as dotted lines. The MVSG for H_{ws} in Figure 3.2 has a node for each committed transaction of H_{ws} (t_1 and t_2) and two edges e: (e_1) from t_1 to t_2 due to $r_1(x) <^H w_2(x)$; (e_2) from t_2 to t_1 due to $r_2(y) <^H w_1(y)$. H_{ws} is not serializable and, thus, the MVSG contains a pivot structure (two consecutive vulnerable edges e_1 and e_2).

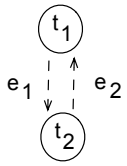


Figure 3.2: MVSG for History H_{ws}

3.2.3 Serializable Snapshot Isolation Protocol

The SSI protocol proposed by Cahill et al. [CRF09] ensures serializability by preventing pivot structures. The main idea is to check SI histories at runtime for structures that can evolve into pivot structures. We call such structures *potential pivot structures*. A potential pivot structure is a pivot structure without the requirement that the three (not necessarily distinct) participating transactions have committed. It evolves into a pivot structure once all participating transactions have committed. The set of transactions in potential pivot structures is naturally a superset of the transactions in pivot structures. For each detected potential pivot structure, one of the participating transactions is aborted to prevent it from evolving into a pivot structure. This approach guarantees that the resulting histories are serializable, but it may produce false positives, i.e., not every potential pivot structure finally results in a non-serializable history. Our implementation leverages this idea and aborts transactions that participate in potential pivot structures (see Section 3.5).

3.3 Modeling Data Relation Snapshots and Defining the Oshiya Scheduling Relation Schemata for DSSI

In order to implement DSSI with Oshiya, we have to (1) specify the schema of the scheduling relations that model the scheduler state (Section 3.3.2), (2) formalize the protocol specification based on these relations (Section 3.4), and (3) implement the protocol specification as scheduling queries (Section 3.5).

For Oshiya, we make the same assumptions and use the same notation as described in Section 2.3.

3.3.1 Modeling Snapshots with Data Relations

Before we specify the scheduling relations for DSSI, we adapt the schemata of the relations addressed by the requests that have to be scheduled. We refer to these relations as *data relations*. We adapt the schemata of the data relations in order to support multiple versions (snapshots) of data items. We model the snapshots explicitly by extending the schemata of data relations. This allows us to achieve database independence and to run DSSI on DBMSs that do not support snapshots. We identify a version of data item x using a tuple (TA, Seq) where TA is the transaction

that created the version and Seq is the position of the request within this transaction. Of course, versions can be modeled differently but this is orthogonal to our approach and beyond the scope of this work. Given a database schema with relations $R_1, \ldots, R_n$, we map each relation R_i to a relation R_i' which has four additional attributes. These attributes model the version identifier for the creator transaction (*CTA* and *CSeq*) and, if applicable, for the transaction that deleted the data item (*DTA* and *DSeq*). The primary key of R_i' is the primary key of R_i union the attributes *CTA* and *CSeq*.

Example 7. Assume a bank stores account data with account numbers and balances in relation *Accounts*(AccNr, Bal). We map this relation to *Accounts'* by extending its schema with the four additional attributes mentioned above. The example instance shown in Figure 3.3 contains an initial version of object x created by transaction t_1 ($CTA = 1, CSeq = 1$) and two new versions created by t_2 ($CTA = 2, CSeq = 2$) and t_3 ($CTA = 3, CSeq = 1$), whereas, relation *Accounts* contains only one version of x (the latest version).

Accounts

AccNr	**Bal**
x	15

Accounts'

AccNr	**Bal**	**CTA**	**CSeq**	**DTA**	**DSeq**
x	5	1	1	-	-
x	10	2	2	-	-
x	15	3	1	-	-

Figure 3.3: Modeling Snapshots with Data Relations

3.3.2 Oshiya Scheduling Relation Schemata

For DSSI, we use the schemata for *scheduling relations* $\mathcal{R}$, $\mathcal{H}$ and $\mathcal{E}$ shown in Figure 3.4. For simplicity, we present only attributes needed for scheduling and omit those necessary for request execution (e.g., the value to be written for write requests).

For each incoming request, we insert a tuple into $\mathcal{R}$ storing an identifier T_i for the transaction t_i that issued the request (*TA*), the request position within this transaction (*Seq*), the type of operation (read, write, abort or commit, modeled by attribute *Op*) and the data object the requests

$\mathcal{R}$ (TA, Seq, Op, Ob)
$\mathcal{H}$ (ID, TA, Seq, Op, Ob, OTA, OSeq)
$\mathcal{E}$ (ID, TA, Seq, Op, Ob, OTA, OSeq)

Figure 3.4: Oshiya Scheduling Relation Schemata

is applied to (*Ob*). Transactions identifiers (*TA*) are ordered, i.e., if $bot_i < bot_j$ then $T_i < T_j$. $\mathcal{H}$ and $\mathcal{E}$ contain additional attributes: ID records the execution order of requests. Together with attribute *Ob*, attributes *OTA* and *OSeq* specify which object version was read by a read request. They correspond to the data relations attributes *CTA* and *CSeq* and are only relevant for read requests. For write, abort, and commit requests these attributes are *null*.

Example 8. Assume the instances of relations $\mathcal{R}$ and $\mathcal{H}$ displayed in Figure 3.5. $\mathcal{H}$ contains the requests that produced the state of relation *Accounts'* from Example 7: (1) and (2) Transaction t_1 created the initial version of object x and committed. (3) Transaction t_2 read this version of object x. (4) and (5) t_2 and t_3 wrote new versions of object x. (6) t_2 committed. (7) t_4 read the new version created by t_2. At this iteration, $\mathcal{R}$ contains no pending requests that have to be scheduled.

$\mathcal{R}$

TA	Seq	Op	Ob

$\mathcal{H}$

ID	TA	Seq	Op	Ob	OTA	OSeq
1	1	1	w	x	-	-
2	1	2	c	-	-	-
3	2	1	r	x	1	1
4	2	2	w	x	-	-
5	3	1	w	x	-	-
6	2	3	c	-	-	-
7	4	1	r	x	2	2

Figure 3.5: Example Instances of Relations $\mathcal{R}$ and $\mathcal{H}$

3.4 DSSI Protocol Specification

We now develop the protocol specification for DSSI based on the scheduling relations presented in Section 3.3. Recall from Chapter 2 that a protocol specification models a protocol as a set of first-order predicate logic expressions over histories.

To formalize SI with Oshiya, we use views over relation $\mathcal{H}$ to get the relevant information described in Section 3.2. For bot, we use view $BOT(TA, ID)$ querying for each transaction (TA) the ID of its first request in $\mathcal{H}$. $EOT(TA, Op, ID)$ selects for each finished transaction t_i (TA) the ID of its final request in $\mathcal{H}$ (corresponds to eot_i) and whether t_i aborted or committed (Op). $Overlap(TA1, TA2)$ contains all pairs of concurrently executed, non-aborted transactions, i.e., they do not have to be committed. *PotPivotStr*$(TA1, TA2, TA3)$ selects all triples of transactions forming potential pivot structures as described in Section 3.2.3.

C1 (Read Versions) The SI protocol specifies [BBG+95, CRF09, WV02] that a read request $r_i(x)$ of a transaction t_i reads t_i's most recent changes to x. If no such changes exist, then $r_i(x)$ reads the latest version of x created by transactions that committed before t_i started. These conditions are formalized as protocol specification constraint C1 (a) and (b) shown in Figure 3.6: **(a)** The first case applies if a transaction T has written object O before reading a version (X, Y) of O:

$$\mathcal{H}(I, T, N, r, O, X, Y) \wedge \mathcal{H}(I_2, T, N_2, w, O, \_, \_) \wedge I_2 < I$$

It follows that T read a version it created itself ($X = T$) and (X, Y) is the latest version produced by T before the read (no newer versions exist):

$$X = T \wedge N_2 = Y \wedge \neg(\mathcal{H}(\_, T, N_2, w, O, \_, \_) \wedge Y < N_2 < N)$$

(b) The second case applies if T has not written O before the read was executed: $\neg(\mathcal{H}(I_2, T, \_, w, O, \_, \_) \wedge I_2 < I)$. It follows that (1) O was written by another transaction X and X committed before T started. (2) (X, Y) has to be the latest version written by X and (3) there may not be another version written by a transaction T_2 that committed after X but before T started:

$$(\mathbf{1})\ X \neq T \wedge EOT(X, c, I_3) \wedge BOT(T, I_4) \wedge I_3 < I_4$$
$$(\mathbf{2})\ \neg(\mathcal{H}(\_, X, N_3, w, O, \_, \_) \wedge N_3 > Y)$$
$$(\mathbf{3})\ \neg(\mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T_2, c, I_5) \wedge I_4 < I_5 < I_3)$$

C2 (FCW) SI requires disjoint write-sets for all committed concurrent transactions. Protocol specification constraint C2 (see Figure 3.6) models this condition as follows. If (1) two overlapping transactions T and T_2 (2) both wrote the same object O and (3) T did already commit, then (4) T_2 did not commit:

$$(\mathbf{1})\ Overlap(T, T_2)$$
$$(\mathbf{2})\ \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_)$$
$$(\mathbf{3})\ EOT(T, c, \_)$$
$$(\mathbf{4})\ \neg EOT(T_2, c, \_)$$

(C1) **(a)** $\forall I, N, O, T, X, Y : \mathcal{H}(I, T, N, r, O, X, Y) \wedge \mathcal{H}(I_2, T, N_2, w, O, \_, \_) \wedge I_2 < I \Rightarrow$
$X = T \wedge N_2 = Y \wedge \neg(\mathcal{H}(\_, T, N_3, w, O, \_, \_) \wedge Y < N_3 < N)$
(b) $\forall I, N, O, T, X, Y : \mathcal{H}(I, T, N, r, O, X, Y) \wedge \neg(\mathcal{H}(I_2, T, \_, w, O, \_, \_) \wedge I_2 < I) \Rightarrow$
$X \neq T \wedge EOT(X, c, I_3) \wedge BOT(T, I_4) \wedge I_3 < I_4 \wedge \neg(\mathcal{H}(\_, X, N_3, w, O, \_, \_) \wedge N_3 > Y) \wedge$
$\neg(\mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T_2, c, I_5) \wedge I_3 < I_5 < I_4)$

(C2) $\forall O, T, T_2 : \; Overlap(T, T_2) \wedge \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T, c, \_)$
$\Rightarrow \neg EOT(T_2, c, \_)$

(C3) $\forall T, T_2, T_3 : \; PotPivotStr(T, T_2, T_3) \Rightarrow \neg(EOT(T, c, \_) \wedge EOT(T_2, c, \_) \wedge EOT(T_3, c, \_))$

Figure 3.6: DSSI Protocol Specification

C3 (Serializability) Recall that an SI history is serializable, if it does not contain pivot structures. In constraint C3 (see Figure 3.6), we follow the approach outlined in Section 3.2.3: If (1) relation $\mathcal{H}$ contains a potential pivot structure, then we require that (2) at least one of the participating transactions did not commit:

$$(\mathbf{1})\ PotPivotStr(T, T_2, T_3)$$
$$(\mathbf{2})\ \neg(EOT(T, c, \_) \wedge EOT(T_2, c, \_) \wedge EOT(T_3, c, \_))$$

3.5 DSSI Implementation

Recall that with Oshiya, protocols are implemented as scheduling queries. We implemented all scheduling queries for DSSI, but herein we only describe $Q_{Schedule}$. Our prototype implementation of Oshiya requires the scheduling queries to be expressed in SQL. However, for conciseness, domain relational calculus expressions are used throughout this section. $Q_{Schedule}$ is developed in two steps. First we present queries necessary to detect potential pivot structures (Section 3.5.1). Afterwards, we use these queries to implement $Q_{Schedule}$ (Section 3.5.2). Recall that detecting potential pivot structures and aborting one of the participating transactions ensure serializability. However, this approach may detect false positives (see Section 3.2). Studying the trade-off be-

tween the number of false positives and the cost of scheduling is an interesting avenue for future work.

3.5.1 Detecting Potential Pivot Structures

We now discuss how to express BOT, EOT, $Overlap$ and $PotPivotStr$ introduced in Section 3.4 as queries over $\mathcal{H}$. BOT and EOT are defined below. E.g., EOT queries for each finished transaction T its abort resp. commit state (A) and its eot (I) which is equal to the ID of its abort resp. commit request in $\mathcal{H}$.

$$\begin{aligned} BOT &= \{T, I \mid \mathcal{H}(I, T, \_, \_, \_, \_, \_) \wedge \neg(\mathcal{H}(I_2, T, \_, \_, \_, \_, \_) \wedge I_2 < I)\} \\ EOT &= \{T, A, I \mid \mathcal{H}(I, T, \_, A, \_, \_, \_) \wedge A = a|c\} \end{aligned}$$

Overlapping transactions are inferred as specified below. Two non-aborted transactions T_1 and T_2 (Equation 3.1) overlap if $bot_1 <^H bot_2$ (Equation 3.2) and $bot_2 <^H c_1$ if T_1 has already committed (Equation 3.3) or the symmetric case (Equation 3.4) holds:

$$\begin{aligned} Overlap = \{T_1, T_2 \mid{}& T_1 \neq T_2 \wedge \neg EOT(T_1|T_2, a, \_) \wedge && (3.1) \\ & ((BOT(T_1, I) \wedge BOT(T_2, I_2) \wedge I < I_2 \wedge && (3.2) \\ & (EOT(T_1, c, I_3) \Rightarrow I_2 < I_3)) \vee && (3.3) \\ & (BOT(T_2, I_2) \wedge BOT(T_1, I) \wedge I_2 < I \wedge (EOT(T_2, c, I_3) => I < I_3)))\} && (3.4) \end{aligned}$$

We use $PotVulnEdge$ to query all potential vulnerable edges between concurrent, non-aborted transactions T and T_2 (potential, because T and T_2 might not yet have committed). $PotPivotStr$ detects potential pivot structures by checking for transactions (T_2) that have both an incoming and outgoing $PotVulnEdge$:

$$\begin{aligned} PotVulnEdge &= \{T, T_2 \mid \mathcal{H}(\_, T, \_, r, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge Overlap(T, T_2)\} \\ PotPivotStr &= \{T, T_2, T_3 \mid PotVulnEdge(T, T_2) \wedge PotVulnEdge(T_2, T_3)\} \end{aligned}$$

Example 9. Consider the history state $\mathcal{H}$ from Example 8 shown in Figure 3.7. For this instance of $\mathcal{H}$, we show the results of the queries defined above (highlighted). For instance, $PotVulnEdge$ contains one potential vulnerable edge from transaction t_2 to t_3, because t_2 and t_3 overlap and t_2 read object x and afterwards t_3 wrote a later version of object x.

$\mathcal{H}$

ID	TA	Seq	Op	Ob	OTA	OSeq
1	1	1	w	x	-	-
2	1	2	c	-	-	-
3	2	1	r	x	1	1
4	2	2	w	x	-	-
5	3	1	w	x	-	-
6	2	3	c	-	-	-
7	4	1	r	x	2	2

BOT

TA	ID
1	1
2	3
3	5
4	7

EOT

TA	Op	ID
1	c	2
2	c	6

Overlap

TA1	TA2
2	3
3	2
3	4
4	3

PotVulnEdge

TAout	TAin
2	3

PotPivotStr

TA1	TA2	TA3

Figure 3.7: Relation $\mathcal{H}$ Containing a Potential Vulnerable Edge

3.5.2 $Q_{Schedule}$

The DSSI version of $Q_{Schedule}$ implementing the protocol specification constraints C1-C3 is shown in Figure 3.8. According to the SI conditions, all write, abort, and read requests from $\mathcal{R}$ may always be selected for execution. $Q_{Schedule}$ selects all of these requests using queries *AbortWrites* and *Reads*. Which commit requests can be selected without violating constraints $C2$ and $C3$ is determined through query *ValidCommits*. In $Q_{Schedule}$, function *GenID()* generates unique values for the *ID* attribute of $\mathcal{H}$ (modeling the execution order of requests).

Read Requests (C1) The $Reads$ query uses *LVV* (last valid version) to select for each read request of transaction T on object O the version (T_2, N_2) that has to be read. Recall that attributes OTA and $OSeq$ of relations $\mathcal{E}$ and $\mathcal{H}$ identify a version of an object O. Version (T_2, N_2) is computed in two steps. *LastOTA* queries the transaction identifier (T_2) of the transaction that wrote the version of O that has to be read by T. Based on this information *LVV* determines N_2, the Seq value of the latest write request of T_2 on object O. T_2 is the maximal value from the following union: (a) T_2=T if T itself created versions of O and (b) transactions that wrote a version of O and committed before T started.

$$(\mathbf{a})\ \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge T = T_2$$
$$(\mathbf{b})\ \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T_2, c, I_2) \wedge (BOT(T, I) \Rightarrow I_2 < I)$$

$$
\begin{array}{rcl}
Q_{Schedule} & = & \{GenID(), T, N, A, O, T_2, N_2 \mid \mathcal{R}(T, N, A, O) \wedge (ValidCommits(T, N, T_2, N_2) \\
 & & \vee AbortsWrites(T, N, T_2, N_2) \vee Reads(T, N, T_2, N_2))\} \\
\hline
AbortsWrites & = & \{T, N, \epsilon, \epsilon \mid \mathcal{R}(T, N, a|w, \_)\} \\
\hline
Reads & = & \{T, N, T_2, N_2 \mid \mathcal{R}(T, N, r, O) \wedge LVV(T, O, T_2, N_2)\} \\
LVV & = & \{T, O, T_2, MAX(N_2) \mid LastOTA(T, O, T_2) \wedge \mathcal{H}(\_, T_2, N_2, w, O, \_, \_)\} \\
LastOTA & = & \{T, O, MAX(T_2) \mid \mathcal{R}(T, \_, r, O) \wedge ((\mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge T = T2) \vee \\
 & & (\mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge EOT(T_2, c, I_2) \wedge (BOT(T, I) \Rightarrow I_2 < I)))\} \\
\hline
ValidCommits & = & \{T, N, \epsilon, \epsilon \mid NonForbCs(T, N) \wedge \neg DelayedCs(T, N)\} \\
DelayedCs & = & \{T, N \mid NonForbCs(T, N) \wedge NonForbCs(T_2, \_) \wedge \\
 & & \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge T > T_2\} \\
NonForbCs & = & \{T, N \mid \mathcal{R}(T, N, c, \_) \wedge \neg(ForbCs(T, N) \vee ForbCinPPS(T, N))\} \\
ForbCinPPS & = & \{T, N \mid \mathcal{R}(T, N, c, \_) \wedge PotPivotStr(T_2, T_3, T_4) \wedge (T = T_2|T_3|T_4) \wedge \\
 & & \neg(\mathcal{R}(T_5, \_, c, \_, \_) \wedge (T_5 = T_2|T_3|T_4) \wedge T < T_5)\} \\
ForbCs & = & \{T, N \mid \mathcal{R}(T, N, c, \_) \wedge \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_) \wedge \\
 & & Overlap(T, T_2) \wedge EOT(T_2, c, \_)\}
\end{array}
$$

Figure 3.8: $Q_{Schedule}$ for DSSI

Example 10. Consider $\mathcal{H}$ from Example 9. $r_2(x)$ read the initial version of object x (since $c_1 <^H bot_2$) and $r_4(x)$ read the version written by t_2 (since $c_2 <^H bot_4$).

Commit Requests (C2 and C3) To guarantee that constraints C2 and C3 hold for each history produced by $Q_{Schedule}$, we have to prevent commit requests to be executed if (1) the commit would violate the FCW rule (C2) or (2) the commit would violate serializability (C3). There are two possible ways how the execution of commit requests can violate the FCW rule: (1a) A commit is from a transaction whose write-set overlaps with the one of a concurrent but already committed transaction and (1b) if $\mathcal{R}$ contains commit requests from multiple transactions with overlapping write-sets, then only one of these transaction may commit. Note that in the concrete implementation, commits identified to violate C2 or C3 are selected by $Q_{Revoked}$ and aborted.

We use a two stage approach to select valid commits: In step 1, query *NonForbCs* selects commits from $\mathcal{R}$ and filters out commits of case 1a using query *ForbCs* and those of case 2 using query *ForbCinPPS*. *NonForbCs* may still contain sets of commit requests from transactions with overlapping write-sets (case 1b). We only allow the oldest transaction from each set to commit. Therefore, in step 2, query *ValidCommits* selects all requests from *NonForbCs* and uses query *DelayedCs* to keep only the commit request of the oldest transaction for each set of transactions

with overlapping write-sets.

Step 1 Query *ForbCs* (case 1a) identifies commits of transactions T that (a) wrote an object also written by an (b) overlapping committed transaction T_2.

$$(\mathbf{a})\ \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_)$$
$$(\mathbf{b})\ \mathit{Overlap}(T, T_2) \wedge \mathit{EOT}(T_2, c, \_)$$

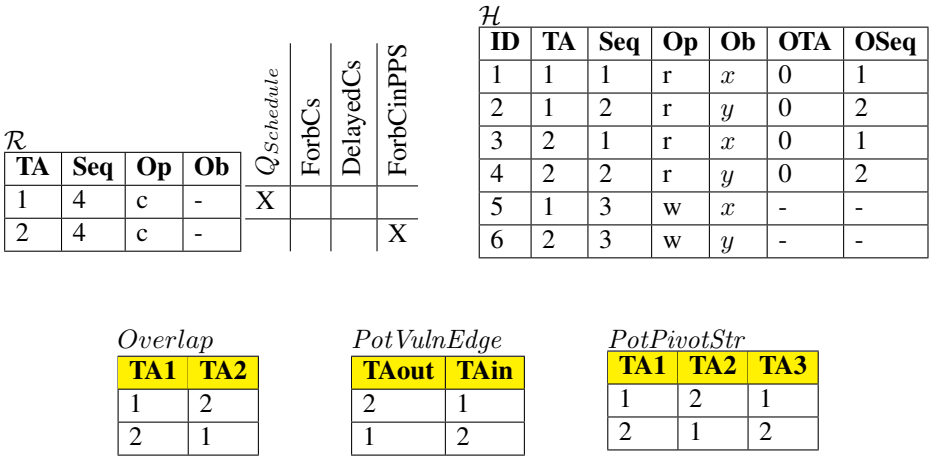
ForbCinPPS (case 2) selects a commit of transaction T from $\mathcal{R}$ if (a) T belongs to potential pivot structure p and (b) $\mathcal{R}$ does not contain a commit request of a younger transaction T_5 (recall that $bot_1 < bot_2 \Rightarrow T_1 < T_2$) also belonging to p. Thus, if $\mathcal{R}$ contains commits of more than one of the transactions belonging to p, we disallow only the youngest one to commit (and abort it using $Q_{Revoked}$).

$$(\mathbf{a})\ \mathit{PotPivotStr}(T_2, T_3, T_4) \wedge (T = T_2|T_3|T_4)$$
$$(\mathbf{b})\ \neg(\mathcal{R}(T_5, \_, c, \_, \_) \wedge (T_5 = T_2|T_3|T_4) \wedge T < T_5)$$

Example 11. Consider the instances of relations $\mathcal{R}$ and $\mathcal{H}$ illustrated in Figure 3.9 that model history H_{ws} from Figure 3.1. To keep the example simple, we do not show the actions of transaction t_0 that created the initial versions of objects x and y. Requests c_1 and c_2 belong to the same potential pivot structure p. Their execution can lead to a write skew violating C3. As shown in Figure 3.9, $Q_{Schedule}$ selects c_1 (smallest TA value). c_2 (commit of youngest transaction) is selected by *ForbCinPPS* and aborted to break p.

Step 2 *DelayedCs* detects case 1b by selecting all transactions T from *NonForbCs* where (a) *NonForbCs* contains another transaction T_2 which (b) wrote an object O that has also been written by T and (c) which is older than T.

$$(\mathbf{a})\ NonForbCs(T_2, \_)$$
$$(\mathbf{b})\ \mathcal{H}(\_, T, \_, w, O, \_, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_, \_)$$
$$(\mathbf{c})\ T > T_2$$



$\mathcal{R}$

TA	Seq	Op	Ob	$Q_{Schedule}$	ForbCs	DelayedCs	ForbCinPPS
1	4	c	-	X			
2	4	c	-				X

$\mathcal{H}$

ID	TA	Seq	Op	Ob	OTA	OSeq
1	1	1	r	x	0	1
2	1	2	r	y	0	2
3	2	1	r	x	0	1
4	2	2	r	y	0	2
5	1	3	w	x	-	-
6	2	3	w	y	-	-

Overlap

TA1	TA2
1	2
2	1

PotVulnEdge

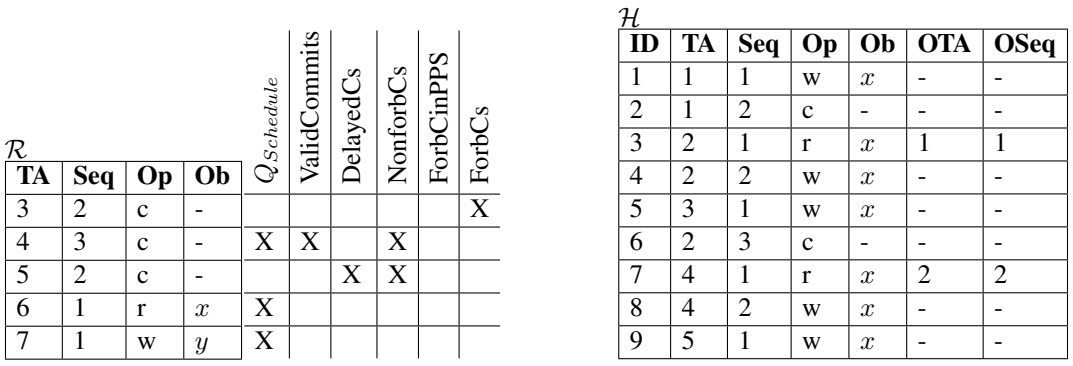
TAout	TAin
2	1
1	2

PotPivotStr

TA1	TA2	TA3
1	2	1
2	1	2

Figure 3.9: Evaluation of $Q_{Schedule}$ for Relation $\mathcal{H}$ Modeling History H_{ws}

Example 12. Consider the instances of $\mathcal{R}$ and $\mathcal{H}$ displayed in Figure 3.10. $Q_{Schedule}$ selects all read ($r_6(x)$) and write ($w_7(y)$) requests. c_3 belongs to *ForbCs* because transaction t_3 wrote the same object as the concurrent but already committed transaction t_2 and is, thus, not allowed to commit. c_4 and c_5 belong to *NonForbCs*, but t_4 and t_5 both wrote the same object x. *ValidCommits* selects only c_4 (oldest transaction from the set $\{t_4, t_5\}$ of transactions with overlapping write-set). c_5 is filtered out by *DelayedCs*.



$\mathcal{R}$

TA	Seq	Op	Ob	$Q_{Schedule}$	ValidCommits	DelayedCs	NonforbCs	ForbCinPPS	ForbCs
3	2	c	-						X
4	3	c	-	X	X		X		
5	2	c	-			X	X		
6	1	r	x	X					
7	1	w	y	X					

$\mathcal{H}$

ID	TA	Seq	Op	Ob	OTA	OSeq
1	1	1	w	x	-	-
2	1	2	c	-	-	-
3	2	1	r	x	1	1
4	2	2	w	x	-	-
5	3	1	w	x	-	-
6	2	3	c	-	-	-
7	4	1	r	x	2	2
8	4	2	w	x	-	-
9	5	1	w	x	-	-

Figure 3.10: Example Evaluation of $Q_{Schedule}$

3.6 Correctness Analysis

We now prove that $Q_{Schedule}$ satisfies serializability constraint C3, denoted as $Q_{Schedule} \models C3$. Recall that an SI history is serializable if it does not contain a pivot structure. Thus, we can

show this fact by proving that $\mathcal{H}$ cannot contain a potential pivot structure between committed transactions (equivalent after Section 3.2.3).

Theorem 3 ($\mathbf{Q_{Schedule}}$ Prevents Pivot Structures)**.** $Q_{Schedule} \models C3$

Proof. We prove Theorem 3 by contradiction. Assume the negation of C3 holds:

$$\begin{aligned} & \neg(\forall T, T_2, T_3: \quad PotPivotStr(T, T_2, T_3) \Rightarrow \neg(EOT(T, c, \_) \wedge EOT(T_2, c, \_) \wedge EOT(T_3, c, \_))) \\ \Leftrightarrow \quad & \exists T, T_2, T_3: \quad PotPivotStr(T, T_2, T_3) \wedge EOT(T, c, \_) \wedge EOT(T_2, c, \_) \wedge EOT(T_3, c, \_) \end{aligned}$$

Let k be the first scheduler iteration where this equation holds for a fixed T_1, T_2, T_3 and T_4.

$$\Leftrightarrow \quad \exists T, T_2, T_3, k: \quad PotPivotStr_k(T, T_2, T_3) \wedge EOT_k(T, c, \_) \wedge EOT_k(T_2, c, \_) \wedge EOT_k(T_3, c, \_)$$

Without loss of generality, let T_3, the transaction at the third position of the potential pivot structure ($PotPivotStr(T, T_2, T_3)$), be the youngest transaction of the participating transactions. This assumption does not result in a loss of generality, because the position of T_3 is irrelevant for the rest of the proof. There must exist a scheduler iteration $i < k$ where T_3 has not yet committed but already belongs to *PotPivotStr*.

$$\Rightarrow \quad \exists i: \quad PotPivotStr_i(T, T_2, T_3) \wedge T_3 > T \wedge T_3 > T_2 \wedge \neg EOT_i(T_3, c, \_)$$

It follows that the commit request c_3 of T_3 occurs in relation $\mathcal{R}$ at some scheduler iteration j ($i < j < k$). To be executed, c_3 has to belong to the set of non-forbidden commits ($NonForbCs$). We can assume $PotPivotStr_i(T, T_2, T_3) \Rightarrow PotPivotStr_j(T, T_2, T_3)$.

$$\Rightarrow \exists j: PotPivotStr_j(T, T_2, T_3) \wedge T_3 > T \wedge T_3 > T_2 \wedge \neg EOT_j(T_3, c, \_) \wedge NonForbCs_j(T_3, \_)$$

We now replace *NonForbCs* by its definition and, afterwards, remove terms that are not needed to derive the contradiction:

$$\begin{aligned}
\Leftrightarrow \quad & \exists j: \quad PotPivotStr_j(T, T_2, T_3) \wedge T_3 > T \wedge T_3 > T_2 \wedge \neg EOT_j(T_3, c, _) \wedge \\
& \mathcal{R}_j(T_3, _, c, _) \wedge \neg ForbCs_j(T_3, _) \wedge \neg ForbCinPPS_j(T_3, _) \\
\Rightarrow \quad & \exists j: \quad PotPivotStr_j(T, T_2, T_3) \wedge T_3 > T \wedge T_3 > T_2 \wedge \mathcal{R}_j(T_3, _, c, _) \wedge \neg ForbCinPPS_j(T_3, _)
\end{aligned}$$

Since c_3 in $\mathcal{R}$ is the commit request of the youngest transaction participating in p, $\mathcal{R}$ cannot contain a commit request of a transaction that is both younger than T_3 and also belongs to p:

$$\begin{aligned}
\Leftrightarrow \quad & \exists j: \quad \mathcal{R}_j(T_3, _, c, _) \wedge PotPivotStr_j(T, T_2, T_3) \wedge \neg(\mathcal{R}_j(T_4, _, c, _) \wedge T_4 = T | T_2 \wedge T_4 < T_3) \wedge \\
& \neg ForbCinPPS_j(T_3, _)
\end{aligned}$$

From the first line of the equation shown above, we can follow $ForbCinPPS_j(T_3, \_)$ which leads to the contradiction and, thus, proves Theorem 3:

$$\Rightarrow \quad \exists j: \quad ForbCinPPS_j(T_3, \_) \wedge \neg ForbCinPPS_j(T_3, \_) \Rightarrow \lightning$$

□

3.7 Related Work

The *ACTA* framework allows to formalize properties of transaction models using first-order formulas over schedules [CR90]. Its conciseness and clarity inspired us to implement schedulers based on declarative protocol specifications. The basic ideas of Oshiya have been presented in [Til10], but this work focused on single-version protocols (2PL) and did not consider correctness. Recent research projects leverage the advantages of declarative languages in various areas [ACC+10, BMK08, CPT+07, KGR+10, WDK+07, YSRG06]. The *Boom* approach uses Overlog to build distributed systems [ACC+10], e.g., a scheduler for MapReduce tasks with policies like First-Come-First-Served. In contrast to our approach, Boom does not focus on database requests or consistency.

Application analysis techniques have been presented in [Fek99, FLO+05] to determine if applications generate serializable executions when running on a system that applies SI. The key idea is that database administrators analyze transaction programs, produce static dependency graphs and manually check for dangerous access patterns leading to non-serializability. Some approaches modify transaction programs to ensure serializable SI schedules: Fekete [FLO+05] proposed the techniques *Materialize* and *Promotion* to achieve serializability. Jorwekar et al. [JFRS07] tried to automate the check whether non-serializable SI executions can occur. However, this approach still requires manual confirmation and modification. Fekete [Fek05] executes certain transactions of pivot structures under S2PL, others run under SI. This approach requires the underlying platform to support both S2PL and SI. Alomari et al. [AFR09] set exclusive locks in an *External Lock Manager* (ELM) to ensure serializability with SI. In contrast to DSSI, these approaches do not work for ad-hoc transactions and require static analysis or manual program modifications.

Another line of work focused on modifying the SI algorithm of the underlying system to ensure serializability. The closest approach to DSSI is the SSI protocol [CRF09] described in Section 3.2.3. This approach modifies the database lock manager with an additional type of locks that are used to detect potential pivot structures. DSSI infers all necessary information to detect and prevent these structures from relation $\mathcal{H}$. Our implementation works with DBMSs out of the box. The underlying DBMS does not even need to provide SI since we model data versions in a standard relational schema (see Section 3.3). Using Oshiya, the implementation of DSSI is close to its formal specification, which enabled us to prove its correctness.

3.8 Conclusions and Future Work

We develop Declarative Serializable Snapshot Isolation (DSSI) using our declarative scheduling model Oshiya. DSSI ensures serializable schedules by avoiding pivot structures and provides database independence. We formally define DSSI as an Oshiya protocol specification, present a scheduler implementation, and prove that the implementation ensures serializability.

In future work, we will experimentally evaluate the performance of DSSI. DSSI ensures serializable schedules by aborting transactions participating in potential pivot structures that can lead to non-serializability, albeit, at the cost of false positives. Investigating the trade-offs involved in reducing the amount of false positives is an interesting avenue for future work.

CHAPTER 4

Resource Acquisition Protocol

Abstract

Booking, reservation, and web shop systems are popular applications where multiple users compete for resources of limited quantity (e.g., flights seats, rooms, or items in stock). This chapter studies the problem of *acquisition process scheduling*, i.e., scheduling concurrent transactions that try to acquire resources. We introduce the *resource acquisition protocol* (RAP), a scheduling protocol that is tailored for acquisition processes. RAP is a deadlock-free protocol that combines the advantages of the snapshot isolation (SI) and two-phase locking protocol (SS2PL), and leverages application semantics to provide low abort rates and low blocking. We define the acquisition graph, a data structure that captures properties of acquisition process schedules and use it to compare the deadlock, abort, and blocking properties of RAP, SS2PL, and SI. Our experiments confirm the analytical results and show that RAP performs better than SS2PL and SI in terms of the number of aborts and blocked transactions.

4.1 Introduction

Scheduling concurrent access to limited resources is a key problem in, e.g., flight and hotel booking, web shops, and ticket reservation systems. A typical usage pattern for these systems is to first browse the available resources, then make a decision on what and how many resources to acquire, and finally acquire the resources. This type of access pattern, which we refer to as an *acquisition process*, leads to a high number of aborts and blocked transactions if the client requests are scheduled using traditional lock-based or multi-versioning protocols. In this work, we introduce a new scheduling protocol (RAP) that addresses these shortcomings and analyze its properties.

S

	ID	**Quant**	**Title**	**Descr**	**Prc**
s_1	u	15	Pericles	Shakespeare, 1607	20
s_2	v	99	Romeo and Juliet	Shakespeare, 1595	30
s_3	w	67	Othello	Shakespeare, 1604	15
s_4	x	70	Hamlet	Shakespeare, 1601	30
s_5	y	16	Macbeth	Shakespeare, 1608	25
s_6	z	34	Timon of Athens	Shakespeare, 1606	20

Table 4.1: Sample Web Shop Database

Example 13. Consider a web shop database storing books in relation S (Table 4.1) where *ID* is a unique item identifier, *Quant* is the available quantity, and *Prc* is the price. For instance, tuple s_1 records item u, a book with title Pericles from Shakespeare with an available quantity of 15. Users Alice, Bob, and Tim browse the web shop searching for books each checking descriptions, prices, and available quantities of several books. Such a process takes several minutes. Alice browses books x, u, w, and y, and orders two copies of books x and y. Bob browses books u and y, and orders two copies of book y. Tim browses books v and x but does not order any of these books. Transactions modeling the behaviour of these users are shown below. Here $r(x)$ checks the availability of a resource (a read request) and $w(x)$ acquires a resource (a write request).

$$\begin{aligned} \text{Tim:} &\ r(v)\ r(x) \\ \text{Bob:} &\ r(u)\ r(y)\ w(y) \\ \text{Alice:} &\ r(x)\ r(u)\ r(w)\ r(y)\ w(x)\ w(y) \end{aligned}$$

Applying generic scheduling protocols to such a scenario is problematic since generic protocols are oblivious to the semantics of the requests, which leads to unnecessary blocking and aborts.

For instance, in an SS2PL schedule, the concurrent execution of Alice and Bob can lead to a deadlock and only one can proceed. This is the case if Alice and Bob both acquired shared locks on y and then requested exclusive locks on y in order to write y. Note that the blocking is not necessary since there are enough copies of the book available to apply the requests of Alice and Bob in any order without violating consistency.

This work addresses the scheduling of acquisition processes $P = \{p_1, \ldots, p_n\}$. Each acquisition process consists of a long resource browsing phase in which resource availability checks are performed on a set of resources $\mathbb{D} = \{d_1, \ldots, d_n\}$, followed by an optional shorter resource acquisition phase in which the selected resources are acquired. We formalize acquisition processes and present the *resource acquisition protocol* (RAP) for scheduling such processes. RAP leverages the advantages of SS2PL and SI, and provides low blocking and low abort rates. Similar to SI browsing customers do not block other customers and are not blocked by other customers. Since browsing transactions do not apply any locks, long browsing phases or inactivity periods of customers during browsing do not affect concurrency. RAP runs updating transactions under strong consistency to ensure consistent resource data. It prevents deadlocks by using solely exclusive locks and by pre-ordering the resources that are acquired. We introduce the *acquisition graph* to analyze schedules of acquisition processes produced by single- and multiversion protocols. The main contributions of this work are:

- We formalize acquisition processes and introduce RAP, a deadlock-free lock-based protocol that leverages the semantics of acquisition processes to provide low aborts rates and ensure low blocking.
- We introduce the acquisition graph, a data structure that allows the analysis and comparison of single- and multiversion protocols for the scheduling of acquisition processes.
- We analyze the deadlock, abort and blocking properties of RAP and compare them to SS2PL and SI. We prove that RAP is deadlock-free; we show that RAP yields fewer histories that are aborted due to insufficient resource availabilities than SI; and we show that RAP results in less blocking than SS2PL.
- Our experiments confirm that RAP performs better than SS2PL and SI with respect to aborts and blocking for various workloads. For the experiments we implemented RAP, SS2PL, and SI using a declarative scheduling model.

The chapter is organized as follows: Section 4.2 formalizes acquisition processes. Section 4.3

introduces the RAP protocol. In Section 4.4 we present the acquisition graph that we use in Section 4.5 to analyze the deadlock, abort and blocking properties of RAP, SS2PL, and SI. Section 4.6 describes the implementation of RAP. Section 4.7 reports experimental results. We discuss related work in Section 4.8 and conclude in Section 4.9.

4.2 Acquisition Process

4.2.1 Preliminaries

A relation schema is denoted as $\mathcal{R} = (A_1, \ldots, A_m)$ where $A_1, \ldots, A_m$ are attributes with domain Ω_i. A tuple r over schema $\mathcal{R}$ is a list containing for every A_i a value $v_i \in \Omega_i$. A relation R over schema $\mathcal{R}$ is a finite set of tuples over $\mathcal{R}$. A *transaction* t_i is a sequence of read $r_i(d)$ and write $w_i(d)$ requests, with d being the accessed data object and i being the transaction identifier, followed by an abort a_i or commit c_i request. $op(s)$ denotes the set of requests contained in a sequence s of requests. The *write-set* $WS(t_i)$ of a transaction t_i is the set of all data items written by t_i: $WS(t_i) = \{d \mid w_i(d) \in op(t_i)\}$. The *read-set* $RS(t_i)$ of a transaction t_i is defined as: $RS(t_i) = \{d \mid r_i(d) \in op(t_i)\}$. We write $\tilde{t}_i$ for a read-only and $\dot{t}_i$ for an updating transaction. The same notation is applied for requests, e.g., $\tilde{c}_i$ denotes a commit of read-only transaction $\tilde{t}_i$.

A *history* h for a set of transactions $T = \{t_1, \ldots t_n\}$ is a sequence of requests which respects the order of requests of each transaction t_i. A schedule (or prefix) s for a history h is a subsequence of h starting at the beginning. $Pref(h)$ denotes the set of all schedules of h. For instance, $s_1 = r_1(x)$ and $s_2 = r_1(x)r_2(x)w_2(x)$ are two possible schedules of history $h = r_1(x)r_2(x)w_2(x)c_2w_1(x)c_2$. We write $p <_s q$ if request p is scheduled before request q according to schedule s. $begin(t_i)$ denotes the start time of transaction t_i and $end(t_i)$ the time when t_i commits or aborts. The execution interval of t_i is $[begin(t_i), end(t_i)]$. Two transactions t_i and t_j are executed *concurrently*, denoted as $Conc(t_i, t_j)$ iff $[begin(t_i), end(t_i)] \cap [begin(t_j), end(t_j)] \neq \emptyset$.

SS2PL: The SS2PL scheduling protocol ensures serializability by using shared read and exclusive write locks. All locks held by a transaction t_i are held until t_i terminates [WV02]. For lock-based protocols, the *block list* for a schedule s contains outstanding requests that are currently blocked because they try to access locked objects, i.e., appending a request from the block list to the schedule s would violate the constraints of the protocol. For instance, assume two transactions $t_1 = r_1(x)w_1(x)c_1$ and $t_2 = r_2(x)r_2(y)c_2$ being scheduled using SS2PL. The block

list for schedule $s = r_1(x)r_2(x)$ is $\{w_1(x)\}$, because t_2 holds a shared-lock on x and, thus, $w_1(x)$ cannot be added to s under SS2PL.

SI: Snapshot Isolation is a multiversion concurrency protocol that maintains multiple versions of data items (tuples) [BBG+95]. Each write $w_i(x)$ creates a new version of item x that is visible to other transactions after c_i. SI ensures that the write set of each pair of concurrent transactions is disjoint. This is ensured by the First-Committer-Wins (FCW) strategy. FCW specifies that a transaction t_i is aborted if at least one item written by t_i has also been written by a concurrent and committed transaction t_j.

4.2.2 Acquisition Processes

Assume a set of *resources* $\mathbb{D} = \{d_1, \ldots, d_m\}$ where each resource has a unique identifier d_k and an available quantity $Q[d_k]$. An acquisition process p_i is a triple $(\tilde{t}_i, \dot{t}_i, AS_i)$ consisting of a *browsing transaction* $\tilde{t}_i$, an *acquisition transaction* $\dot{t}_i$, and an *acquisition set* AS_i. Browsing transaction $\tilde{t}_i$ models the long resource browsing phase during which resource availability checks are performed, e.g., browsing items in a web shop. $\tilde{t}_i$ is a read-only transaction that models availability checks as requests $\tilde{r}_i(d)$. During browsing a subset of the browsed resources is selected for acquisition and added to acquisition set AS_i, e.g., a shopping cart: $AS_i \subseteq RS(\tilde{t}_i)$. Acquisition transaction $\dot{t}_i$ models the shorter resource acquisition phase during which the resources of the acquisition set are acquired. $\dot{t}_i$ is only executed if $AS_i \neq \emptyset$, e.g., customers can only submit an order if at least one item is in the shopping cart. Resource acquisition requests are modeled as $\dot{r}_i(d)$ and $\dot{w}_i(d)$ requests. An acquisition transaction $\dot{t}_i$ does not perform blind writes and $RS(\dot{t}_i) = WS(\dot{t}_i) = AS_i$ holds. $Q[d, \dot{t}_i]$ denotes that $\dot{t}_i$ plans to acquire resource d with quantity $Q[d, \dot{t}_i]$. $Q[\dot{r}_i(d)]$ resp. $Q[\dot{w}_i(d)]$ specifies the quantity of d read by request $\dot{r}_i(d)$ resp. written by $\dot{w}_i(d)$. We require that the value $Q[\dot{w}_i(d)]$ written by a write request $\dot{w}_i(d)$ only depends on value $Q[\dot{r}_i(d)]$ read by $\dot{r}_i(d)$. $\dot{t}_i$ aborts if resources are not available in sufficient quantity: $\exists d \in WS(\dot{t}_i) : Q[\dot{r}_i(d)] < Q[d, \dot{t}_i]$.

A history for a set of acquisition processes $P = \{p_1, \ldots, p_n\}$ is a history for the transactions in P with the property that for each process $p_i \in P$ all requests of the browsing transaction of p_i are executed before any request of the acquisition transaction of p_i.

Example 14. The web shop scenario can be modeled as three acquisition processes for Alice $p_1 = (\tilde{t}_1, \dot{t}_1, \{x, y\})$, Bob $p_2 = (\tilde{t}_2, \dot{t}_2, \{y\})$, and Tim $p_3 = (\tilde{t}_3, \dot{t}_3, \emptyset)$ accessing a set of resources $\mathbb{D} = \{u, v, w, x, y, z\}$:

$$
\begin{aligned}
\mathbf{p_1}: \quad & \tilde{t}_1 = \tilde{r}_1(x)\ \tilde{r}_1(u)\ \tilde{r}_1(w)\ \tilde{r}_1(y)\ \tilde{c}_1 \\
& \dot{t}_1 = \dot{r}_1(x)\ \dot{w}_1(x)\ \dot{r}_1(y)\ \dot{w}_1(y)\ \dot{c}_1 \\
\mathbf{p_2}: \quad & \tilde{t}_2 = \tilde{r}_2(u)\ \tilde{r}_2(y)\ \tilde{c}_2 \\
& \dot{t}_2 = \dot{r}_2(y)\ \dot{w}_2(y)\ \dot{c}_2 \\
\mathbf{p_3}: \quad & \tilde{t}_3 = \tilde{r}_3(v)\ \tilde{r}_3(x)\ \tilde{c}_3
\end{aligned}
$$

Let s be a sequence of requests and $T(s)$ be the set of all transactions participating in s. Aborted, browsing, and acquisition transactions in s are defined as:

$$
\begin{aligned}
A(s) &= \{t_i \mid t_i \in T(s) \wedge a_i \in op(s)\} \\
\tilde{T}(s) &= \{t_i \mid t_i \in T(s) \wedge WS(t_i) = \emptyset\} \\
\dot{T}(s) &= \{t_i \mid t_i \in T(s) \wedge WS(t_i) \neq \emptyset\}
\end{aligned}
$$

Consider the processes $p_2 = (\tilde{t}_2, \dot{t}_2, \{y\})$ and $p_3 = (\tilde{t}_3, \dot{t}_3, \emptyset)$, and schedule s shown below. The following holds: $A(s) = \emptyset$, $T(s) = \{\tilde{t}_2, \dot{t}_2, \tilde{t}_3, \dot{t}_3\}$, $\tilde{T}(s) = \{\tilde{t}_2, \tilde{t}_3\}$, and $\dot{T}(s) = \{\dot{t}_2, \dot{t}_3\}$.

$$s = \tilde{r}_2(u)\tilde{r}_2(y)\tilde{r}_3(v)\tilde{c}_2\dot{r}_2(y)\tilde{r}_3(x)$$

4.3 RAP

This section introduces the resource acquisition protocol (RAP), a protocol for scheduling acquisition processes as defined in Section 4.2. RAP is a lock-based protocol that limits blocking by allowing browsing transactions to perform dirty reads. In contrast to SS2PL and SI, which have to abort transactions due to protocol constraints, no such aborts occur in RAP schedules. Under SS2PL, deadlocks have to be resolved by aborting transactions. SI aborts a transaction if it wrote at least one resource that has been written by a concurrently executed transaction that already committed.

Definition 7. (*RAP*) Let $\mathbb{D} = \{d_1, \ldots, d_n\}$ be a set of resources, Ξ be a total order over $\mathbb{D}$, and let $d <_\Xi d'$ denote that d is smaller than d' according to order Ξ. A history h for a set of acquisition processes $P = \{p_1, \ldots, p_n\}$ is a RAP history if the following conditions hold:

$$
\begin{aligned}
(1)\ & \forall \dot{t}_i, \dot{t}_j \in T(h), d \in \mathbb{D} : Conc(\dot{t}_i, \dot{t}_j) \wedge \dot{r}_i(d) \\
& \Rightarrow \neg(\dot{r}_i(d) <_h \dot{r}_j(d) <_h \dot{c}_i) \wedge \neg(\dot{r}_i(d) <_h \dot{r}_j(d) <_h \dot{a}_i) \wedge \\
& \neg(\dot{r}_i(d) <_h \dot{w}_j(d) <_h \dot{c}_i) \wedge \neg(\dot{r}_i(d) <_h \dot{w}_j(d) <_h \dot{a}_i) \\
(2)\ & \forall \dot{t}_i \in T(h), d, d' \in \mathbb{D} : \dot{r}_i(d) <_h \dot{r}_i(d') \Rightarrow d <_\Xi d'
\end{aligned}
$$

Condition 1 The first condition guarantees that an acquisition transaction $\dot{t}_j$ may not access a resource d that has already been accessed by a concurrent acquisition transaction $\dot{t}_i$ until $\dot{t}_i$ is finished. Disallowing concurrent acquisition transactions on the same resource guarantees that the following anomalies cannot occur: (a) negative resource availability values, i.e., customers ordered more resources than available, (b) resource quantities that do not match the actual available quantities. RAP realizes the first condition using exclusive locks. Acquisition transactions lock each resource exclusively before reading it. The exclusive locks of a transaction $\dot{t}_i$ are held until $\dot{t}_i$ terminates.

Similar to SI, browsing reads are never blocked by other requests and do not block other requests in RAP schedules. A browsing transaction $\tilde{t}_i$ is allowed to read uncommitted changes of a resource d made by a concurrent acquisition transaction. Hence, long browsing phases or inactivity periods of customers do not affect concurrency. Note that consistency is not affected by the dirty reads, because an acquisition transaction is required to re-check the availability of a resource (read request) before acquiring it. It is possible that an acquisition transaction aborts due to insufficient resource availability, although, during browsing the resource was available in sufficient quantity. This is a consequence of the given setting where acquisition processes are modeled as browsing transactions and acquisition transactions and not an RAP specific problem. In Section 4.5.2 we show that under RAP such aborts occur in less schedules than with SI. This is because browsing transactions always see up-to-date values instead of outdated snapshots.

		Scheduled		
		$\tilde{\mathbf{r}}_\mathbf{i}(\mathbf{d})$	$\dot{\mathbf{r}}_\mathbf{i}(\mathbf{d})$	$\dot{\mathbf{w}}_\mathbf{i}(\mathbf{d})$
Requested	$\tilde{\mathbf{r}}_\mathbf{j}(\mathbf{d})$	+	+	+
	$\dot{\mathbf{r}}_\mathbf{j}(\mathbf{d})$	+	-	-
	$\dot{\mathbf{w}}_\mathbf{j}(\mathbf{d})$	+	-	-

Table 4.2: Compatibility of Requests for RAP

Table 4.2 shows the compatibility of request types under RAP. Given an executed request of a non-finished transaction t_i, Table 4.2 shows which types of pending requests of a concurrent transitions t_j are permitted (+) or blocked (-). For instance, if an acquisition transaction $\dot{t}_i$ has

$h = \tilde{r}_1(y)\ \tilde{r}_3(v)\ \tilde{r}_1(u)\ \tilde{r}_2(u)\ \tilde{r}_1(w)\ \tilde{r}_2(y)\ \tilde{r}_1(x)\ \tilde{c}_2\tilde{c}_1\ \dot{r}_1(x)\ \tilde{r}_3(x)\ \dot{w}_1(x)\ \dot{r}_2(y)\ \dot{w}_2(y)\ \tilde{c}_3\ \dot{c}_2\ \dot{r}_1(y)\ \dot{w}_1(y)\dot{c}_1$

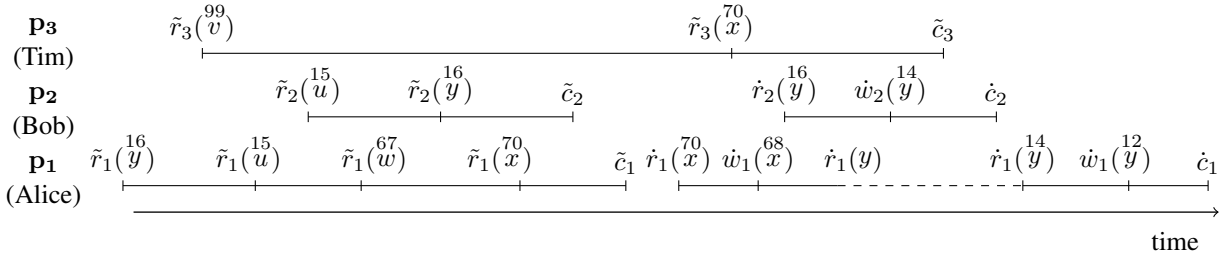


Figure 4.1: Example RAP History for the Web Shop Scenario

executed a read on a resource d then no concurrent acquisition transaction $\dot{t}_j$ is allowed to execute any request on d.

Condition 2 The second condition requires that the order of the resources that are acquired by an acquisition transaction complies with a total order Ξ over the set of resources $\mathbb{D}$. This is feasible since the acquisition set AS_i, i.e., the resources selected for acquisition during browsing transaction $\tilde{t}_i$, is known by the application before $\dot{t}_i$ is executed. Hence, the application can simply order the acquisitions inside $\dot{t}_i$ according to order Ξ beforehand. The total order over the acquisition requests [HD91] applied by RAP in combination with using a single type of exclusive lock is sufficient to guarantee deadlock-free schedules as we will show in Section 4.5.1.

Example 15. Consider the RAP history of the processes of Alice ($p_1 = (\tilde{t}_1, \dot{t}_1, \{x, y\})$), Bob ($p_2 = (\tilde{t}_2, \dot{t}_2, \{y\})$), and Tim ($p_3 = (\tilde{t}_3, \dot{t}_3, \emptyset)$) given in Figure 4.1. The acquisition transactions of Alice and Bob are executed concurrently. Since Bob's read $\dot{r}_2(y)$ exclusively locks y, the acquisition read $\dot{r}_1(y)$ of Alice on y is blocked until Bob finishes his acquisition. Note that SS2PL and SI would abort the acquisition of either Alice or Bob, SS2PL because of a deadlock between both transactions and SI because of their overlapping write-set. Tim is allowed to perform the read on the exclusively locked resource x, because RAP never blocks browsing reads.

4.4 Acquisition Graph

In this section, we introduce the *acquisition graph*, a tool that enables us to analyze single- and multiversion protocols when scheduling acquisition processes according to their blocking and abort behaviour. We use the acquisition graph in Section 4.5 to compare RAP with SS2PL and SI. The acquisition graph for a schedule s is a directed graph. Nodes represent non-aborted

transactions. An edge from transaction t_i to t_j indicates their concurrent execution, and edge labels denote dependencies between concurrent transactions.

Definition 8. (*Acquisition Graph*) Let $\mathbb{D} = \{d_1, \ldots, d_n\}$ be a set of resources, s be a schedule and b be the block list of s. The acquisition graph $AG(s) = (V, E, L)$ of s is defined as:

$$V = \{t_i \mid t_i \in (T(s) - A(s))\}$$
$$E = \{(t_i, t_j) \mid t_i, t_j \in V \wedge Conc(t_i, t_j)\}$$

L is a function $L : E \rightarrow Pow(\mathbb{L})$ that annotates each edge in E with a set of labels from a domain $\mathbb{L}$: $\mathbb{L} = \bigcup_{d \in \mathbb{D}} \{\tilde{r}\dot{w}(d), \dot{w}\tilde{r}(d), \dot{r}\dot{r}(d), ah(d), ch(d)\}$. For a given edge $e = (t_i, t_j)$ the value $L(e)$ is defined as:

$$\begin{aligned}
\tilde{r}\dot{w}(d) \in L(e) &\Leftrightarrow t_i \in \tilde{T}(s) \wedge t_j \in \dot{T}(s) \wedge \tilde{r}_i(d) <_s \dot{w}_j(d),\\
\dot{w}\tilde{r}(d) \in L(e) &\Leftrightarrow t_i \in \dot{T}(s) \wedge t_j \in \tilde{T}(s) \wedge \dot{w}_i(d) <_s \tilde{r}_j(d),\\
\dot{r}\dot{r}(d) \in L(e) &\Leftrightarrow t_i, t_j \in \dot{T}(s) \wedge \dot{r}_i(d) \in (op(s) \cup op(b)) \wedge\\
&\qquad \dot{r}_j(d) \in (op(s) \cup op(b)),\\
ah(d) \in L(e) &\Leftrightarrow t_j \in \dot{T}(s) \wedge \dot{w}_j(d) \in op(s) \wedge\\
&\qquad (\dot{r}_i(d) \in op(b) \vee \tilde{r}_i(d) \in op(b) \vee \dot{w}_i(d) \in op(b)),\\
ch(d) \in L(e) &\Leftrightarrow \dot{r}_i(d) \in op(s) \wedge\\
&\qquad (\tilde{r}_j(d) \in op(s) \vee \dot{r}_j(d) \in op(s)) \wedge\\
&\qquad \dot{w}_i(d) \in op(b).
\end{aligned}$$

The edge-labels of an acquisition graph denote dependencies between transactions that enable us to reason about properties of acquisition process schedules. The interpretations of these labels is as follows: If two transactions $\tilde{t}_i$ and $\dot{t}_j$ participating in schedule s are executed concurrently and $\tilde{t}_i$ read a resource d before $\dot{t}_j$ wrote d, then $AG(s)$ contains an edge $(\tilde{t}_i, \dot{t}_j)$ with label $\tilde{r}\dot{w}(d)$ denoting this $\tilde{r}\dot{w}$-dependency (analog for $\dot{w}\tilde{r}(d)$). Edge $(\tilde{t}_i, \dot{t}_j)$ is labeled $\dot{r}\dot{r}(d)$ if transactions $\tilde{t}_i$ and $\dot{t}_j$ both read or intend to read resource d. Label $\dot{r}\dot{r}$ is the only label that always occurs symmetrically. All other labels are unidirectional. The ah- and ch-dependencies denote that a transaction is waiting for another transaction. If transaction t_i attempts to *acquire* a lock on resource d but transaction t_j already *holds* an exclusive lock on d and, thus, t_i waits for t_j, then we say transaction t_i is in state *lock acquisition wait*. This corresponds to an edge (t_i, t_j) that is labeled with $ah(d)$. If transactions t_i and t_j *hold* shared locks on d and t_i attempts to *convert* its shared lock to an exclusive lock and, thus, waits for t_j, then we say transaction t_i is in state *lock*

conversion wait. This corresponds to an edge (t_i, t_j) with label $ch(d)$.

Example 16. Assume Bob browses y while Alice acquires y. Figure 4.2 shows a possible SS2PL schedule s for their acquisition processes $p_1 = (\tilde{t}_1, \dot{t}_1, \{y\})$ with $\tilde{t}_1 = \tilde{r}_1(y)\tilde{c}_1$, $\dot{t}_1 = \dot{r}_1(y)\dot{w}_1(y)\dot{c}_1$ and $p_2 = (\tilde{t}_2, \dot{t}_2, \{y\})$ with $\tilde{t}_2 = \tilde{r}_2(y)\tilde{c}_2$, $\dot{t}_2 = \dot{r}_2(y)\dot{w}_2(y)\dot{c}_2$. The requests in the block list of s ($\dot{w}_1(y)$ and $\dot{w}_2(y)$) are marked by angle brackets. Dashed lines indicate lock waits. Dotted lines denote dependencies that are recorded as edge-labels in the acquisition graph. $AG(s)$ (Figure 4.2) contains a node for every non-aborted transaction in s: $\tilde{t}_1$, $\tilde{t}_2$, $\dot{t}_1$ and $\dot{t}_2$. Transactions $\dot{t}_1$ and $\tilde{t}_2$ as well as $\dot{t}_1$ and $\dot{t}_2$ have been executed concurrently, thus, edges are drawn between these pairs of transactions. Edges $(\dot{t}_1, \dot{t}_2)$ and $(\dot{t}_2, \dot{t}_1)$ are labeled with $\dot{r}\dot{r}(y)$ since $\dot{t}_1$ and $\dot{t}_2$ both read y. Edge $(\dot{t}_1, \dot{t}_2)$ is labeled with $ch(y)$ because $\dot{t}_1$ attempts to convert its shared lock on y into an exclusive lock but is blocked by $\dot{t}_2$ which also holds a shared lock on y (analog for edge $(\dot{t}_2, \dot{t}_1)$).

Given an edge e labeled with a set $L(e) = \{l_1, \ldots, l_n\}, 1 \leq i \leq n$ of labels $l_i = g_i(m_i)$ with g_i being the label type and m_i being a resource. The *label reduce operator* α eliminates the resource of each label l_i such that $\alpha(L(e)) = \{g_1, \ldots, g_n\}$. A *path* p is a set of directed edges $p = \{e_1, \ldots, e_n\}$ with $e_i \in E$ and for each $e_i = (t_j, t_k)$ and $e_{i+1} = (t_l, t_m)$ holds that $t_k = t_l$. A path p is called a *cycle* if $e_1 = (t_i, t_j)$ and $e_n = (t_k, t_l)$ with $t_i = t_l$. The intersection set $\mathcal{L}(p)$ of edge labels of a path p is defined as $\mathcal{L}(p) = \bigcap_{e \in p} L(e)$. The intersection set $\bar{\mathcal{L}}(p)$ of edge labels of a path p that ignores the resource attached to an edge label is defined as $\bar{\mathcal{L}}(p) = \bigcap_{e \in p} \alpha(L(e))$.

Example 17. Consider the path $p = \{e_1, e_2\}$ with $e_1 = (t_1, t_2)$, $e_2 = (t_2, t_3)$, $L(e_1) = \{ah(x)\}$, and $L(e_2) = \{ah(y)\}$. Then $\mathcal{L}(p) = \emptyset$ and $\bar{\mathcal{L}}(p) = \{ah\}$.

4.5 Analysis of RAP

In this section, we leverage the acquisition graph introduced in the previous section to analyze the deadlock, abort, and blocking properties of RAP, SS2PL, and SI. Furthermore, we prove that RAP is deadlock-free. We establish conditions when RAP, SS2PL and SI produce aborts due to insufficient resource availability and prove that with RAP less histories lead to such aborts than with SI. We show that, in contrast to SS2PL, RAP completely avoids blocking between a browsing transaction and a browsing or acquisition transaction. However, RAP applies more rigorous blocking for acquisition transactions which is necessary to prevent deadlocks based on lock conversion. Our experimental evaluation (Section 4.7) demonstrates that the negative effect on performance caused by rigorous blocking of acquisition transactions is insignificant.

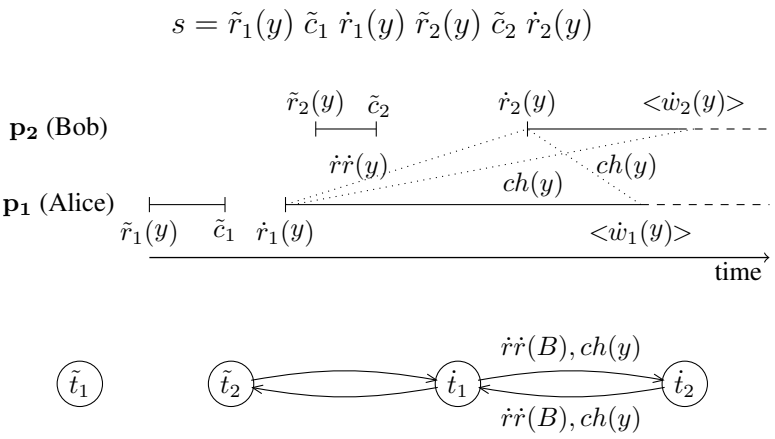


Figure 4.2: SS2PL Example History and Corresponding Acquisition Graph

4.5.1 Deadlocks

An aborted acquisition process is an unwanted situation. Users spent a long time for browsing resources, e.g., searching for books, reading sample pages. Not being able to acquire the resources that have been selected during browsing is a frustrating experience. As commonly known, deadlocks can occur under SS2PL (cf. Example 16) and have to be resolved by aborting participating transactions. A transaction is aborted by SI if its write-set overlaps with a concurrently executed transaction that already committed. We refer to this types of aborts as *protocol aborts*, because they are caused by constraints of the protocol. Although, RAP is a lock-based protocol, it does not generate schedules with deadlocks, in other words, RAP is deadlock-free. This is because (1) RAP uses solely exclusive locks and, thus, no lock conversion waits can occur. (2) RAP leverages the fact that the acquisition set for an acquisition transaction is known beforehand. This means we can pre-order the resource acquisitions according to a global order Ξ to avoid lock acquisition wait cycles (see Def. 7) that lead to deadlocks.

Theorem 4. *Let H_{RAP} be the set of histories valid under RAP and DL(s) denote that schedule s contains a deadlock. The following holds:*

$$\forall h \in H_{RAP} : \forall s \in Pref(h) : \neg DL(s)$$

Proof. Deadlocks due to lock conversion waits (cf. Example 16) cannot occur since RAP only uses one type of locks. It remains to show that lock acquisition waits cannot lead to deadlocks under RAP. A deadlock caused by lock acquisition waits would occur in a RAP schedule s if

$AG(s)$ contains a cycle c with $ah \in \bar{\mathcal{L}}(c)$, i.e., a cycle where each transaction in the cycle attempts to acquire a lock that is currently held by another transaction in the cycle (ah-cycle). We prove the theorem by contradiction. Assume that the acquisition graph $AG(s)$ for an RAP schedule s contains an ah-cycle. We will show that such a schedule violates Condition 2 of the RAP protocol. Let s be such a schedule and $\{d_1, \ldots, d_n\}$ denote the resources the transactions in the ah-cycle are waiting for. Since the cycle is of finite length and only one transaction can hold a lock on an item at a time ($\forall i, j \in \{1, \ldots, n\} : d_i = d_j \Rightarrow i = j$), we know that one resource d_i in this set is greater than all other resources in the set according to the order Ξ: $\forall j \neq i : d_j <_\Xi d_i$. Let $\dot{t}_i$ be the transaction that is waiting to acquire a lock on d_i and $\dot{t}_j$ be the transaction that is currently holding the lock on d_i. Since $\dot{t}_j$ is part of the cycle it is currently waiting to acquire a lock on a resource d_j. Thus, $\dot{t}_j$ has accessed d_i before attempting to access d_j. However, $d_j <_\Xi d_i$ holds which contradicts with the assumption that $\dot{t}_j$ accesses all resources in ascending order according to Ξ (Condition 2). □

4.5.2 Aborts

The available quantity of a resource d is checked twice in an acquisition process p_i: Once in the browsing transaction ($\tilde{r}_i(d)$) and once in the acquisition transaction before the resource is acquired ($\dot{r}_i(d)$). These two reads may observe different quantities for the same resource, if the quantity was modified by a concurrent process between the two read requests. Thus, an acquisition transaction $\dot{t}_i$ may have to abort because $\dot{r}_i(d)$ reveals that the quantity of d is insufficient ($Q[\dot{r}_i(d)] < Q[d, \dot{t}_i]$), although, during browsing the quantity was sufficient ($Q[\tilde{r}_i(d)] \geq Q[d, \dot{t}_i]$). We only consider situations where transactions that access the same object are executed concurrently. This is because aborts due to insufficient quantities cannot be prevented by any protocol if these transactions are executed serially. We call such aborts *availability aborts*.

Definition 9. (*Availability Abort*) Let d be a resource, s be a schedule, and processes $p_i = (\tilde{t}_i, \dot{t}_i, AS_i)$, $p_j = (\tilde{t}_j, \dot{t}_j, AS_j)$ participate in s. An abort $\dot{a}_i \in op(s)$ is called an availability abort if $\dot{t}_j$, that has been executed concurrently with $\tilde{t}_i$, changed $Q[d]$ so that $Q[\tilde{r}_i(d)] \geq Q[d, \dot{t}_i] \wedge Q[\dot{r}_i(d)] < Q[d, \dot{t}_i]$.

To compare the availability abort behavior of RAP to SS2PL and SI, we establish sufficient and necessary conditions for an availability abort to occur in a schedule under each of these protocols. When RAP or SS2PL is applied, an availability abort of an acquisition process p_i can only occur if at least one concurrent process p_j acquired the same resource between the browsing read $\tilde{r}_i(d)$

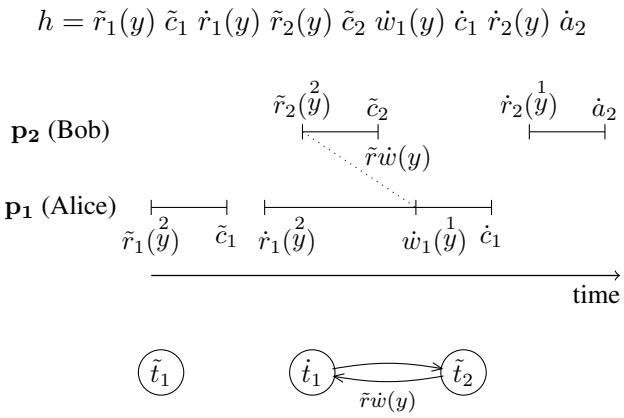


Figure 4.3: SI/SS2PL/RAP History with Availability Abort and Corresponding Acquisition Graph

and acquisition read $\dot{r}_i(d)$. This is reflected in the acquisition graph as $\tilde{r}\dot{w}$-dependency, i.e., an edge e with $\tilde{r}\dot{w} \in \bar{\mathcal{L}}(e)$. If such an edge exists, an availability abort of $\dot{t}_i$ occurs only if the quantity of d read during $\dot{t}_i$ is insufficient: $Q[\dot{r}_i(d)] < Q[d, \dot{t}_i]$. We deduce the following condition. Let $\mathbb{D} = \{d_1, \ldots, d_n\}$ be a set of resources. For RAP resp. SS2PL, a history h contains an availability abort, denoted as $AA(h, RAP)$ resp. $AA(h, SS2PL)$, iff

$$\exists s \in Pref(h), (\tilde{t}_i, \dot{t}_j) \in AG(s), d \in \mathbb{D}:$$
$$\tilde{r}\dot{w}(d) \in L((\tilde{t}_i, \dot{t}_j)) \wedge Q[\tilde{r}_i(d)] \geq Q[d, \dot{t}_i] \wedge Q[\dot{r}_i(d)] < Q[d, \dot{t}_i]$$

Example 18. Assume resource y has a quantity of $Q[y] = 2$. Bob browses y and decides to acquire y with a quantity of $Q[y, \dot{t}_2] = 2$. Concurrently, Alice acquires y with a quantity of $Q[y, \dot{t}_1] = 1$. History h shown in Figure 4.3 illustrates this situation. Note that this history is valid for all three protocols considered in this work. Although, Bob read a quantity $Q[\tilde{r}_2(y)] = 2$ during browsing, his acquisition is aborted due to insufficient availability of y. This is because Alice wrote $Q[\dot{w}_1(y)] = 1$ between the two reads of Bob. The dependency between transactions $\tilde{t}_2$ and $\dot{t}_1$ which caused the availability abort of $\dot{t}_2$ is reflected in $AG(h)$ as an $\tilde{r}\dot{w}$-dependency.

Analog to SS2PL and RAP, $\tilde{r}\dot{w}$-dependencies can lead to availability abort under SI (Figure 4.3). Additionally, with SI an availability abort of an acquisition transaction $\dot{t}_i$ can occur if the quantity

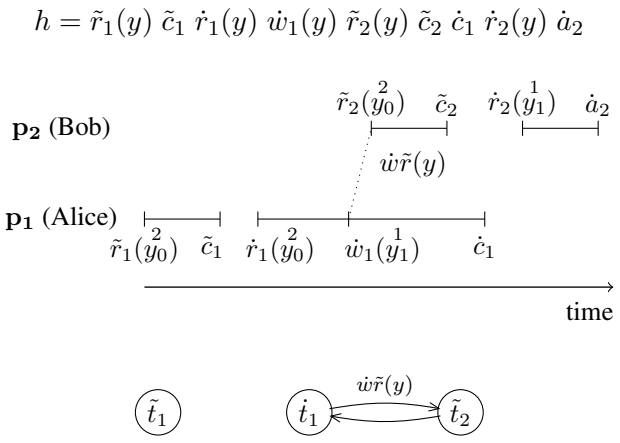


Figure 4.4: SI History with Availability Abort and Corresponding Acquisition Graph

of d that was read during browsing was already out-dated, i.e., the decision to acquire d was based on an out-dated quantity. This is due to the fact that a browsing transaction $\tilde{t}_i$ executed with SI only sees the last snapshots of resources that were committed before $\tilde{t}_i$ started. Such a situation occurs if $AG(s)$ contains an $\dot{w}\tilde{r}$-dependency, i.e., an edge e with $\dot{w}\tilde{r} \in \bar{\mathcal{L}}(e)$. Thus, an SI history h contains an availability abort, denoted as $AA(h, SI)$, iff

$$\begin{aligned}
&\exists s \in Pref(h), (\tilde{t}_i, \dot{t}_j) \in AG(s), (\dot{t}_j, \tilde{t}_i) \in AG(s), d \in \mathbb{D}: \\
&(\tilde{r}\dot{w}(d) \in L((\tilde{t}_i, \dot{t}_j)) \vee \dot{w}\tilde{r}(d) \in L((\dot{t}_j, \tilde{t}_i)))\wedge \\
&Q[\tilde{r}_i(d)] \geq Q[d, \dot{t}_i] \wedge Q[\dot{r}_i(d)] < Q[d, \dot{t}_i]
\end{aligned}$$

Example 19. ($\dot{w}\tilde{r}$*-dependency*) Assume resource y has a quantity of $Q[y] = 2$. SI history h shown in Figure 4.4 captures a situation where Bob browses y and decides to acquire y with a quantity of $Q[y, \dot{t}_2] = 2$ while Alice concurrently acquires y with a quantity of $Q[y, \dot{t}_1] = 1$. Bob's acquisition is aborted because during browsing he read an out-dated quantity $Q[\tilde{r}_2(y_0)] = 2$. This is because under SI Bob read version y_0 that was valid before $\tilde{t}_2$ started and not the updated uncommitted version y_1 created by Alice's acquisition. In $AG(h)$ this is represented as a $\dot{w}\tilde{r}$-dependency.

RAP and SS2PL have the same condition for producing availability aborts. We now prove that RAP generates availability aborts in less histories than SI.

Theorem 5. *Let $\mathbb{H}$ be the set of histories that are valid under RAP and SI. The following holds:*

$$\forall h \in \mathbb{H} : AA(h, RAP) \Rightarrow AA(h, SI) \tag{4.1}$$

$$\exists h \in \mathbb{H} : AA(h, SI) \not\Rightarrow AA(h, RAP) \tag{4.2}$$

Proof. Proving (1): We substitute $AA(h, RAP)$ and $AA(h, SI)$ by their definitions.

$$\begin{aligned}
&\forall h \in \mathbb{H} : \\
&(\exists s \in Pref(h), (\tilde{t}_i, \dot{t}_j) \in AG(s), (\dot{t}_j, \tilde{t}_i) \in AG(s), d \in \mathbb{D} : \\
&\tilde{r}\dot{w}(d) \in L((\tilde{t}_i, \dot{t}_j)) \wedge \\
&Q[\tilde{r}_i(d)] \geq Q[d, \dot{t}_i] \wedge Q[\dot{r}_i(d)] < Q[d, \dot{t}_i]) \Rightarrow \\
&(\exists s \in Pref(h), (\tilde{t}_i, \dot{t}_j) \in AG(s), (\dot{t}_j, \tilde{t}_i) \in AG(s), d \in \mathbb{D} : \\
&(\tilde{r}\dot{w}(d) \in L((\tilde{t}_i, \dot{t}_j)) \vee \dot{w}\tilde{r}(d) \in L((\dot{t}_j, \tilde{t}_i))) \wedge \\
&Q[\tilde{r}_i(d)] \geq Q[d, \dot{t}_i] \wedge Q[\dot{r}_i(d)] < Q[d, \dot{t}_i])
\end{aligned}$$

This formula is of the form:

$$\begin{aligned}
&\forall h \in \mathbb{H} : \exists x : (a(x) \wedge c(x)) \\
&\Rightarrow \exists x, y : ((a(x) \vee b(x, y)) \wedge c(x))
\end{aligned}$$

Due to distributivity of conjunction we get the tautology:

$$\begin{aligned}
\Leftrightarrow &\forall h \in \mathbb{H} : \exists x : (a(x) \wedge c(x)) \\
&\Rightarrow \exists x, y : ((a(x) \wedge c(x)) \vee (b(x, y) \wedge c(x))).
\end{aligned}$$

Proving (2): We now prove the existence of a schedule s with the property that all extensions of s to a history under SI contain an availability abort, but one extension of s to a history under RAP does not contain an availability abort. Consider the following schedule s that is both a valid RAP and SI schedule. Schedule s is a prefix of the history given in Example 19.

$$s = \tilde{r}_1(y)\ \tilde{c}_1\ \dot{r}_1(y)\ \dot{w}_1(y)\ \tilde{r}_2(y)\ \tilde{c}_2\ \dot{c}_1\ \dot{r}_2(y)$$

Under SI, Bob browses a quantity of $Q[\tilde{r}_2(y_0)] = 2$ and decides to acquire y with a quantity

of $Q[y, \dot{t}_2] = 2$. During acquisition, $\tilde{r}_2(y_1)$ returns $Q[\tilde{r}_2(y_1)] = 1$ and, thus, his acquisition is aborted because $Q[y, \dot{t}_2] > Q[\dot{r}_2(y_1)] = 1$. This yields the history shown in Figure 4.4 with $AA(h, SI)$. Note that for p_1 and p_2 this is the only valid SI extension of s.

Scheduling the same schedule with RAP, Bob browses the already updated quantity $Q[\tilde{r}_2(y)] = 1$. Thus, he decides to acquire y with a quantity of $Q[y, \dot{t}_2] = 1$. During acquisition, $\dot{r}_2(y)$ still returns $Q[\dot{r}_2(y)] = 1$ and, thus, his acquisition can commit. This results in the history given in Figure 4.4 with the exception that Bob commits his acquisition ($\dot{c}_i$) and, thus, $\neg AA(h, RAP)$ holds. □

4.5.3 Blocking

We now analyze the blocking behavior of RAP and compare it to SS2PL and SI. The blocking characteristics of the protocol that is applied to acquisition processes can be evaluated using the acquisition graph. Given a schedule s and its acquisition graph $AG(s)$, edge labels ah and ch indicate that a transaction $t_i \in T(s)$ is currently blocked. An edge e with $ch \in \bar{\mathcal{L}}(e)$ indicates that an acquisition transaction is in state lock conversion wait. A lock conversion wait occurs if a transaction $\tilde{t}_i$ resp. $\dot{t}_i$ and a concurrent transaction $\dot{t}_j$ both read resource d and now $\dot{t}_j$ intends to write d. An edge e with $ah \in \bar{\mathcal{L}}(e)$ indicates that a transaction is in state lock acquisition wait. This state occurs if transaction $\tilde{t}_j$ resp. $\dot{t}_j$ cannot read resource d because a concurrent acquisition transaction $\dot{t}_i$ is holding an exclusive lock on d.

Table 4.3 shows which request combinations lead to ah and ch labels under which protocol. The requests are assumed to access the same resource d and the request of process p_i is to be considered as already scheduled, the one of p_j as requested. The remaining combinations do not lead to blocking under any of the protocols because shared locks are compatible ($\tilde{r}_i\tilde{r}_j$, $\tilde{r}_i\dot{r}_j$, $\dot{r}_i\tilde{r}_j$) or cannot occur since acquisition transactions do not perform blind writes ($\dot{w}_i\dot{w}_j$).

SI With SI (FCW) no transaction is ever blocked and, thus, labels ah and ch do not occur in ac-

	1 $\tilde{\mathbf{r}}_\mathbf{i}\dot{\mathbf{w}}_\mathbf{j}$	2 $\dot{\mathbf{r}}_\mathbf{i}\dot{\mathbf{r}}_\mathbf{j}$	3 $\dot{\mathbf{r}}_\mathbf{i}\dot{\mathbf{w}}_\mathbf{j}$	4 $\dot{\mathbf{w}}_\mathbf{i}\tilde{\mathbf{r}}_\mathbf{j}$	5 $\dot{\mathbf{w}}_\mathbf{i}\dot{\mathbf{r}}_\mathbf{j}$
RAP		ah			ah
SS2PL	ch		ch	ah	ah
SI					

Table 4.3: Blocking Behavior of RAP, SS2PL and SI

quisition graphs of SI schedules. This results in high concurrency but comes at the cost of aborts due to insufficient resource availability and due to overlapping write-sets (aborted to ensure data consistency).

SS2PL SS2PL applies shared locks for browsing transactions and shared and exclusive locks for acquisition transactions, thus, resulting extensive blocking and deadlocks (for Combinations 1, 3-5). Long running browsing transactions can block acquisition transactions (Combination 1) which negatively affects concurrency.

RAP RAP blocks only if necessary. Long running browsing transactions are not blocked and do not block ensuring high concurrency (Combinations 1,4). Knowing that every $\dot{w}_i(d)$ request has a preceding $\dot{r}_i(d)$ request, RAP applies exclusive locks already for acquisition reads disallowing concurrent accesses on the same resource and, thus, ensure consistency without introducing deadlocks. An acquisition write $\dot{w}_j(d)$ can never be blocked by an acquisition read $\dot{r}_i(d)$ (Combination 3) under RAP, because the preceding $\dot{r}_j(d)$ request would have been blocked by the exclusive lock on d.

In summary, in contrast to SS2PL, RAP avoids blocking of browsing transactions and blocking of acquisition transactions by browsing transactions. For acquisition transactions RAP directly applies exclusive locks to avoid blocking and deadlocks due to lock conversion waits.

4.6 Implementation

We implemented RAP, SS2PL, and SI using the Oshiya scheduling model [TGBK11a, TGBK11b], because this model enables the rapid development of protocols and allows us to study these protocols on a single platform. In Oshiya, the scheduler state is represented as *scheduling relations*: pending requests in relation $\mathcal{R}$ and requests that have already been scheduled in relation $\mathcal{H}$. The requests in relation $\mathcal{H}$ represent the request history (schedule) generated so far, including the execution order. The protocol logic is implemented by so-called scheduling queries $Q_{Schedule}$, $Q_{Revoked}$ and $Q_{Irrelevant}$. In this section, we limit the discussion to $Q_{Schedule}$. See [TGBK11a] for an in-depth discussion. Scheduling is performed iteratively, in *scheduling iterations*, by executing $Q_{Schedule}$ to retrieve sets of pending requests from $\mathcal{R}$ and adding them to the history ($\mathcal{H}$).

Scheduling Relations Figure 4.5 shows the scheduling relations $\mathcal{R}$ and $\mathcal{H}$ for RAP. Attribute

$\mathcal{R}$

TA	TT	Seq	Op	Ob	Val
3	ta	3	r	y	-
4	ta	2	w	y	12
5	tb	1	r	x	-

$\mathcal{H}$

ID	TA	TT	Seq	Op	Ob	Val
1	1	tb	1	r	x	-
2	1	tb	2	r	y	-
3	2	tb	1	r	y	-
4	1	tb	3	c	-	-
5	2	tb	2	c	-	-
6	3	ta	1	r	x	-
7	3	ta	2	w	x	5
8	4	ta	1	r	y	-

$\mathcal{X}$

Ob	TA
x	3
y	4

Figure 4.5: Example Execution of $Q_{Schedule}$

ID models the order of requests in the schedule (only in $\mathcal{H}$), TA is the transaction identifier, TT is the transaction type ('tb' for browsing and 'ta' for acquisition transactions), Seq is the relative position of the request within the transaction, Op denotes the request type, Ob the object, and Val is the value to be written for write requests. For example, the read request $\tilde{r}_1(x)$, the first request of browsing transaction $\tilde{t}_1$, is modeled as tuple $(1, 1, tb, 1, r, x, -)$ in relation $\mathcal{H}$.

$Q_{Schedule}$ We implement the first RAP condition (ref. Def. 7) using locks. An acquisition transaction $\dot{t}_i$ that read an object O holds an exclusive lock on O until it terminates. Traditional implementations of lock-based protocols use a lock table to store which locks are held at a certain point in time (*explicit locking*). We use *implicit locking*, a different locking approach that infers the locks held by transactions from the available request history ($\mathcal{H}$). RAP solely uses exclusive locks. These exclusive locks are inferred from $\mathcal{H}$ by query $\mathcal{X}$, given below as a relational calculus expression. We denoted variables by capital letters, constants by small letters, and unrestricted variables by '_'. Non-target variables are implicitly existentially quantified. E.g., for $\{A \mid \exists B : I(A, B, C) \wedge (C = \text{'}r\text{'} \vee C = \text{'}w\text{'})\}$ we write $\{A \mid I(A, B, r|w)\}$.

$$\mathcal{X} = \{O, T \mid \mathcal{H}(\_, T, ta, \_, r, O, \_) \wedge \neg\mathcal{H}(\_, T, \_, \_, a|c, \_, \_)\}$$

If an acquisition transaction $\dot{t}_i$ read object O ($\mathcal{H}(\_, T, ta, \_, r, O, \_)$), then it holds an exclusive lock on O until it finishes ($\mathcal{H}(\_, T, \_, \_, a|c, \_, \_)$). Note that this means that the lock actually appears one scheduler iteration after the write request has been selected from $\mathcal{R}$. This requires that we ensure that only one acquisition transaction accesses O during a single scheduling iteration. As an example for the evaluation of query $\mathcal{X}$, consider the instance of relation $\mathcal{H}$ and the result of query $\mathcal{X}$ shown in Figure 4.5. Acquisition transactions $\dot{t}_3$ and $\dot{t}_4$ are (logically) locking resources x and y exclusively.

$$
\begin{aligned}
Q_{Schedule} = \; & \{GenID(), T, Y, N, A, O, V \mid \mathcal{R}(T, Y, N, A, O, V) \wedge \\
& (OnePerObj(\_, T) \vee A = a|c \vee TBReads(T, O))\} \\
OnePerObj = \; & \{O, T \mid LegalOps(T, O) \wedge \\
& \neg LegalOps(T_2, O) \wedge T_2 < T\} \\
LegalOps = \; & \{T, O \mid \mathcal{R}(T, \_, \_, \_, O, \_) \wedge \\
& \neg XBlocked(T, O) \wedge \neg TBReads(T, O)\} \\
TBReads = \; & \{T, O \mid \mathcal{R}(T, tb, \_, r, O, \_)\} \\
XBlocked = \; & \{T, O \mid \mathcal{R}(T, \_, \_, \_, O, \_) \wedge \mathcal{X}(O, T_2) \wedge T \neq T_2\}
\end{aligned}
$$

Figure 4.6: $Q_{Schedule}$ implementing RAP

Using $\mathcal{X}$, we can implement the RAP protocol as query $Q_{Schedule}$ given in Figure 4.6. In subquery $LegalOps$, we take all requests from $\mathcal{R}$ and subtract two sets of requests: Requests on objects that are exclusively-locked by another transaction ($XBlocked$) and browsing reads which are handled separately ($TBReads$). To cover the case where multiple pending requests try to access the same unlocked object, we use subquery $OnePerObj$ to select one request per object from $LegalOps$. In $Q_{Schedule}$, we select the requests identified by $OnePerObj$ as well as all abort and commit requests ($OnePerObj(\_, T) \vee A = a|c$). In addition, $Q_{Schedule}$ selects all browsing reads for execution ($TBReads(T, O)$) because their execution is not restricted by the protocol. The *GenID()* function generates unique increasing numerical identifiers used to establish a total request order in $\mathcal{H}$.

Example 20. Consider the scheduling relation state shown in Figure 4.5. The content of $\mathcal{H}$ is interpreted as follows: Browsing transactions $\tilde{t}_1$ and $\tilde{t}_2$ read resources x and y. Both transactions already committed. The subsequent acquisition transaction $\dot{t}_3$ acquired x. Acquisition transaction $\dot{t}_4$ read y. $\mathcal{R}$ contains three pending requests: $\dot{t}_3$ attempts to read y, $\dot{t}_4$ plans to update y, and $\tilde{t}_5$ attempts to browse x. For this instance of $\mathcal{H}$ the subquery $\mathcal{X}$ infers two locks: resources x and y are exclusively locked by transactions $\dot{t}_3$ and $\dot{t}_4$. $Q_{Schedule}$ selects the following two requests from $\mathcal{R}$ for execution (yellow highlighted): The update request of $\dot{t}_4$ is selected for execution since $\dot{t}_4$ is holding the lock on y. Despite of object x being exclusively locked by $\dot{t}_3$, $Q_{Schedule}$ selects $\tilde{r}_5(x)$ for execution. The acquisition read $\tilde{r}_3(y)$ is not selected, because y is exclusively locked by $\dot{t}_4$.

4.7 Experiments

In the experimental evaluation, we compare the performance of Oshiya implementations of RAP, SS2PL [TGBK10], and SI [TGBK11a] when used to schedule different acquisition process workloads. The main goal is to extend the theoretical qualitative analysis of Section 4.5 with a quantitative analysis.

Methodology We simulated 1000 concurrent clients, each executing acquisition processes continuously. An experiment is finished when 4000 resources have been acquired successfully. In each experiment, we use the same workload for all protocols and measure the number of aborts and the throughput as committed acquisition processes per scheduling iteration.

Clients acquire resources with a quantity of $Q_{AT} = 5$. The objects to browse are randomly selected (Gaussian distribution). During a browsing transaction $\tilde{t}_i$, at most Ob_{AT} of the Ob_{BT} browsed resources are randomly picked for acquisition, i.e., a resource d is selected for acquisition if $Q[\dot{r}_i(d)] \geq Q_{AT}$ and the number of already selected resources is less than Ob_{AT}. The pre-ordering of resources is applied for all protocols. A process refrains from acquiring a resource d if the quantity of d observed by the browsing read is lower than Q_{AT}. Between each browsing read the client waits for D_{BT} scheduling iterations before executing the next read. This parameter is used to control the length of browsing transactions. We assigned each of the 1000 clients a browsing delay between 20 and 50 (randomly chosen). The experiment parameters are described in Table 4.4.

Parameter	Symbol
#resources browsed	Ob_{BT}
#resources acquired	Ob_{AT}
Delay between browsing reads	D_{BT}
Initial resource quantity	Q_{Init}
Database size	DBS

Table 4.4: Experiment Parameters

Setup We used two machines with a 2.8GHz single-core CPU with 2GB memory running Ubuntu 8.10 resp. MS Windows Server 2003. The first simulates the clients. The second runs the backend DBMS and Oshiya scheduler [TGBK11b]. For SS2PL, we used a deadlock resolution technique that detects cycles and aborts one participating transaction from each cycle. For SI we apply the FCW strategy.

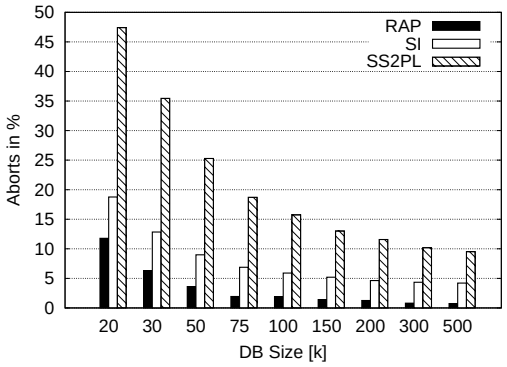
4.7.1 Varying Conflict Probability

We first analyze the influence of the conflict probability on aborts and blocking. We vary the *database size* from $DBS = 20k$ (high conflict probability) to $DBS = 500k$ (low conflict probability) while fixing $Q_{Init} = 20$, $Ob_{BT} = 5$, $Ob_{AT} = 3$. We expect SS2PL and SI to perform best for low conflict probability because the probability of deadlocks and overlapping write-sets is low, and to perform worst for high conflict probability ($DBS = 20k$). As shown in Figure 4.7(a), for high conflict probability ($DBS = 20k$), RAP aborts 11.8% of the started acquisition processes, whereas, SS2PL aborts 47.4% and SI 18.8%. For low conflict probability ($DBS = 500k$) the abort rates are 9.5% (SS2PL), 4.2% (SI), and only 0.7% (RAP). All aborts for RAP are availability aborts. In addition to availability aborts, SI produces aborts due to overlapping write-sets, and SS2PL due to deadlocks.

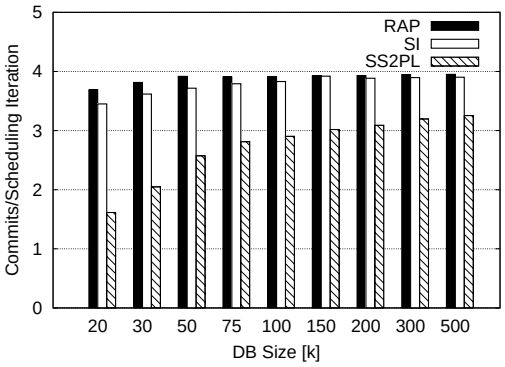
Crucial for SS2PL are the shared locks held by long running browsing transactions causing long term blocking and deadlocks between browsing and acquisition transactions. Thus, SS2PL provides the lowest throughput (Figure 4.7(b)), has the longest process durations (Figure 4.7(d)), and requires more scheduling iterations (Figure 4.7(c)) to acquire the 4k resources than SI or RAP. For low conflict probability ($DBS = 500k$), RAP achieves a throughput roughly equal to the one of SI (3.9 commits/scheduling iteration). For high conflict probability ($DBS = 20k$), RAP has a throughput that is 7% higher than the one of SI. Transactions can be blocked under RAP, but only for the (short) duration of an acquisition transaction (Figure 4.7(d)). Under SI progress is delayed, because more transactions are aborted due to overlapping write-sets.

4.7.2 Varying the Number of Acquisitions

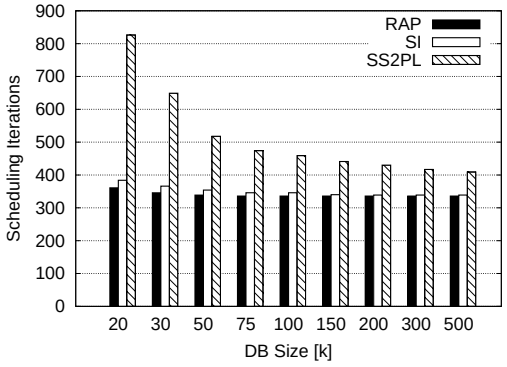
To analyze the influence of the ratio between reads and writes we vary the number of acquired resources ($Ob_{AT} = 1, 3, 5, 7, 10$) while fixing $DBS = 100k$, $Q_{Init} = 20$, $Ob_{BT} = 10$. RAP has the lowest abort rate for all tested parameter values (cf. Figure 4.8(a)). The abort rate of SI and SS2PL strongly increases for more write-intensive workloads. This is because the probability of overlapping write-sets resp. deadlocks increases in the number of acquired resources. As expected, the throughput decreases when the number of abort increases (compare Figure 4.8(a) and Figure 4.8(b)). SI is expected to perform better for read-intensive workloads. However, even for such workloads, RAP performs as good as SI (Figure 4.8(b)). The reason is that RAP solely blocks acquisition transactions which is of no consequence for read-intensive workloads. For



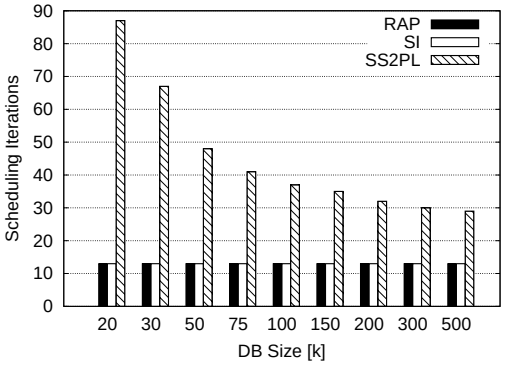
(a) Abort Rate



(b) Throughput



(c) Scheduling Iterations



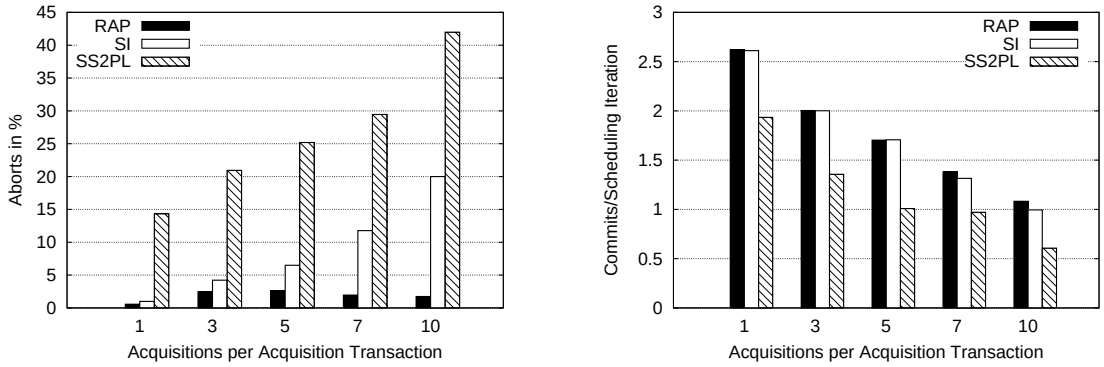
(d) Average Process Duration

Figure 4.7: Varying Conflict Probability

write-intensive workloads, RAP performs slightly better than SI. The blocking of RAP has less impact on performance than the increasing number of aborts of SI. Again, SS2PL provides the lowest throughput due to extensive blocking and aborts.

4.7.3 Varying Resource Availability

In this experiment we vary the initial resource availability ($Q_{Init} = 15, 20, 30, 40, 70, 100$) while fixing $DBS = 30k$, $Ob_{BT} = 5$, $Ob_{AT} = 5$. All resources are set to an initial quantity of Q_{Init} and are not restocked during the experiment; every protocol has to deal with the same lack of resources. We expect an increasing number of availability aborts when decreasing the initial



(a) Abort Rate

(b) Throughput

Figure 4.8: Varying the Number of Acquisitions

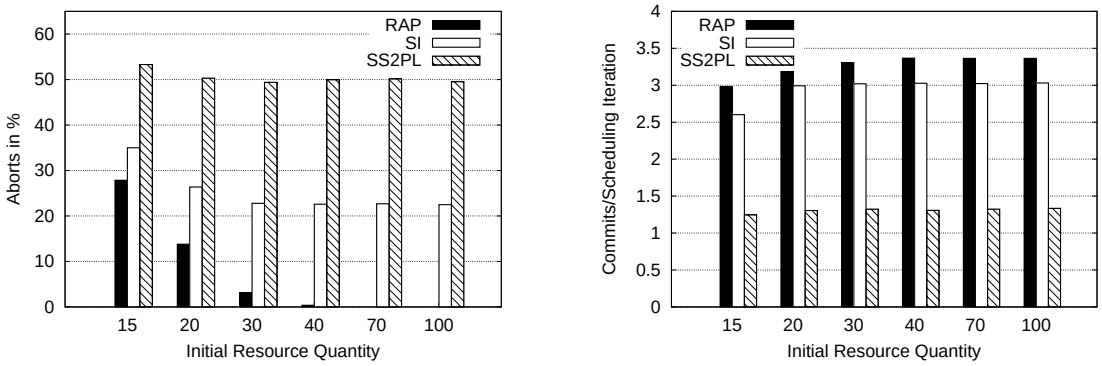
resource quantity. RAP has the lowest abort rate for all quantities as shown in Figure 4.9(a) and does not abort any transactions for initial resource quantities larger than 40. In contrast, SI resp. SS2PL produce 22.6% resp. 50% aborts for these parameter values. As shown in Figure 4.9(b), the throughput of RAP is about 6-14% resp. 136-157% higher than the one of SI resp. SS2PL.

4.7.4 Summary

The experimental evaluation confirms our theoretical results. RAP produces less aborts than SS2PL and SI for all workloads while maintaining a higher throughput than SS2PL and an equal or slightly higher throughput than SI. If resources are available in sufficient quantity, then RAP does not abort any acquisition processes. The throughput of RAP and SI is mainly determined by the read/write ratio. For SS2PL, the throughput is additionally heavily influenced by the conflict probability. The abort rate of RAP is mainly determined by resource availability; the one of SS2PL and SI by database size and read/write ratio.

4.8 Related Work

Three lines of work are related to our approach: Domain specific scheduling protocols, solutions for long-running transactions, and data structures that are used to analyze the properties of scheduling protocols.



(a) Abort Rate

(b) Throughput

Figure 4.9: Varying Initial Resource Availability

Domain Specific Scheduling An extensive amount of work adds additional scheduling layers on top of standard DBMSs to overcome limitations of DBMS scheduling in terms of QoS support [BF99, KKDK07, Sch06b], or scalability and performance [AABM08, CMZ04, ENTZ04, PA04]. The state-of-the-art is to develop domain-specific protocols for a given application [CdMP08, CRS+08, DHJ+07, KHAK09, BFG+06, TGBK11b]. Seifert et al. [SS03] and Bernstein et al. [BFG+06] allow transactions to read out-of-date information (data currency) to increase concurrency. Kraska et al. [KHAK09] propose Consistency Rationing for cloud environments that allows to define consistency guarantees on data instead of transactions, and to automatically switch consistency at runtime. Halici et al. [HD91] presented ODL, an optimistic locking technique for distributed databases that similar to RAP leverages a-priori knowledge to acquire locks in a predefined order to prevent deadlocks. In contrast to these approaches, we develop a protocol for a type of transactions (resource acquisitions) that to the best of our knowledge has not been studied by previous work. RAP outperforms traditional protocols for this setting without sacrificing consistency.

Long-Running Transactions Long-running transactions can seriously impact performance for lock-based protocols (long-term blocking) and are likely to abort under SI (high probability of overlapping write-sets). Solutions comprise approaches that split such transactions in small nested subtransactions using predefined compensating transactions [CR90, GMS87] to compensate for failed subtransactions and approaches that require additional application knowledge to guarantee serializability over subtransactions [SLSV95, CdMP08]. Chianese et al. [CdMP08] developed a method to improve the concurrency of long-running transactions by pre-serializing

conflicting transactions in a middleware layer based on application semantics. Yalamanchi et al. [YG09] propose compensation-aware data types as a solution for long running business transactions. This approach uses data-type specific atomic update and undo operations that commute with each other. Thus, the order of operations can be relaxed without sacrificing consistency. We deal with long-running read-only transactions by avoiding locks at the cost of dirty reads. However, dirty reads are unproblematic for resource acquisition processes, because acquisitions transactions are required to recheck resource availability before executing a write.

Data-structures for Analyzing Protocols Data structures for analyzing the properties of schedules are usually tailored for specific protocol types (e.g., lock-based protocols). For instance, the waits-for graph, conflict graph [WV02], multiversion serialization graph [CRF09], or dependency serialization graph [FLO+05] are used to determine serializability or deadlocks for either locking or multiversion protocols. We introduce the acquisition graph, because none of these structures can be used to analyze acquisition process schedules of single- and multiversion protocols with respect to availability aborts, protocol aborts, and blocking.

4.9 Conclusions and Future Work

We address the problem of scheduling resource acquisitions processes and introduce the resource acquisition protocol (RAP), a protocol that is specifically tailored for scheduling such processes. RAP leverages application semantics to provide low abort rates and low blocking. We analyze and compare the deadlock, abort, and blocking properties of RAP, SS2PL, and SI. Our experiments confirm that RAP performs better than SS2PL and SI with respect to aborts and blocking.

In future work, we will investigate techniques for reducing the number of availability aborts or let processes fail early-on instead of during the acquisition transaction. One promising approach is to return fake quantities of zero if the quantity of a resource falls below a threshold. This would cause clients to fail during browsing if the probability of a later availability abort is high, i.e., we avoid long-running transactions that are bound to fail at the cost of potentially unnecessary aborts.

CHAPTER 5

The Oshiya Debugger and Analyzer

Abstract

This chapter presents the Oshiya Debugger and Analyzer (ODA), a novel tool for debugging, visualizing, and comparing scheduling protocols developed using the Oshiya declarative scheduling model. ODA supports typical debugging features such as navigation, e.g., stepping through a protocol execution, and a declarative variant of break points. In addition, we support comparison of protocols through simultaneous execution over the same workload as well as visual analysis of protocol execution statistics through a feature called statistic queries. ODA extends standard debugging features by, e.g., stepping backward through a protocol execution and automatically providing context information for break points. We demonstrate the features of the system by comparing the Oshiya implementations of two example protocols: Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI). A banking example is used to illustrate how Snapshot Isolation (SI) leads to data constraint violations and how SSI prevents such violations by detecting potential pivot structures (PPS) and aborting one of the participating transactions.

5.1 Introduction

This chapter presents the **O**shiya **D**ebugger and **A**nalyzer (ODA), a tool for debugging, visualizing, and comparing scheduling protocols developed using the Oshiya declarative scheduling model [TGBK11a]. We demonstrate the features of ODA by comparing the Oshiya implementations of two example protocols: Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI). A banking example is used to illustrate how Snapshot Isolation (SI) leads to data constraint violations and how SSI prevents such violations by detecting potential pivot structures (PPSs) and aborting one of the participating transactions.

ODA simulates the execution of scheduling protocols over user-provided workloads. In Oshiya, the scheduler state is represented as *scheduling relations*: pending requests in relation $\mathcal{R}$, requests that were chosen for execution in relation $\mathcal{E}$, and requests that have been executed in relation $\mathcal{H}$. The requests in relation $\mathcal{H}$ represent the history (schedule) generated so far, including their execution order. Scheduling is performed iteratively, in *scheduling iterations*, by retrieving and moving sets of requests from $\mathcal{R}$ to $\mathcal{E}$ and from $\mathcal{E}$ to $\mathcal{H}$, according to the protocol implementation [TGBK11a, TGBK11b]. ODA is a novel approach and is, to the best of our knowledge, the first tool that supports the development, debugging, visualization and comparison of scheduling protocols. It offers the following key features:

Interactive Protocol Comparisons ODA supports the simultaneous execution of multiple scheduling protocols over the same workload. This feature is used to compare protocols, e.g., to investigate the handling of PPSs by, respectively, SI and SSI.

Break and Analyze Queries ODA models breakpoints as *break queries* that are executed after each scheduler iteration. Scheduling stops when a break query returns a non-empty result. The results of a break query are visualized by highlighting matching tuples in the scheduling state. Matching tuples are identified through *analyze queries*. The combination of break and analyze queries allows to easily detect and analyze errors in protocol executions and provides an understanding of how a protocol behaves for a certain workload. We illustrate break and analyze queries that detect constraint violations under SI and that identify PPSs in SI and SSI schedules.

Navigational Debugging After a break query stopped the scheduling of requests, the user can debug a protocol implementation by navigating through the execution steps (backward and

forward). For each step, ODA shows the current scheduling and database state. We use navigational debugging to analyze how PPSs are handled under SI and SSI.

Statistical Protocol Analysis ODA provides statistics about protocol executions, and it allows users to register new measures and customize how they are displayed, e.g., as a graph or as tabular data. Custom measures are modeled as *statistic queries* that access the scheduling state, database state, and results of break or analyze queries. ODA collects, aggregates, and visualizes the results of these queries over time. We illustrate this by collecting statistics about the number of additional aborts under SSI to prevent the constraint violations that are possible under SI.

Leveraging the advantages of declarative languages for monitoring, testing, and debugging purposes has recently gained attention in distributed systems research. Gunawi et al. [GDJ+11] use declarative testing specifications to debug errors in distributed systems. We also use declarative queries (relational calculus, SQL) for the analysis, but focus on protocols for scheduling transactions. Furthermore, ODA allows navigational debugging and supports protocol comparisons by executing several protocols in parallel over the same workload.

The chapter proceeds as follows. Section 5.2 sets up our example scenario. In Section 5.3, we describe how ODA supports protocol comparison by the simultaneous execution of multiple protocols. Section 5.4 introduces break and analyze queries, which we use to illustrate that no constraint violations occur under SSI. In Section 5.5, we leverage ODA's navigational controls, in combination with break queries, for detecting PPSs and for analyzing how they are broken by SSI to prevent constraint violations. In Section 5.6, we use the ODA statistics feature to measure the number of additional aborts incurred by preventing constraint violations.

5.2 Example Scenario

The scenario used in this chapter is a comparison and debugging of Oshiya implementations of SI and SSI [TGBK11a]. We first describe a simplified version of the workload that is used to compare the protocols. Next, we describe the relevant parts of SI, SSI, and PPSs.

Example 21. Consider a bank that maintains a relation $Accounts$ as shown in Table 5.1. Acc is the unique account number, Bal is the account balance, $Type$ denotes the account type, and

$Owner$ denotes the account owner. For instance, tuple a_3 records checking account x and tuple a_4 records savings account y, each with a balance of 50 and both belonging to Alice and Bob (owner AB). According to their contract with the bank, the combined balance of their accounts has to be positive, i.e., there exists a constraint $C : x + y \geq 0$. Alice uses an ATM to withdraw \$70 from account x and Bob withdraws \$70 from account y using another ATM. Before each payment, the ATM reads the balances of x and y and checks if C remains satisfied if the withdrawal is executed. The withdrawals of Alice and Bob can be modeled as transactions $t_1 = r_1(x)r_1(y)w_1(x)$ and $t_2 = r_2(x)r_2(y)w_2(y)$. We use the standard notation for read $r_i(x)$ and write $w_i(x)$ requests of a transaction: i is the transaction identifier and x the accessed data item.

Accounts

	Acc	**Bal**	**Type**	**Owner**
a_1	v	50	checking	D
a_2	w	50	saving	D
a_3	x	50	checking	AB
a_4	y	50	saving	AB

Table 5.1: Sample Database for Banking Example

SI [BBG+95] is a multi version concurrency protocol that avoids reads being blocked by writes and vice versa. This is achieved by providing each transaction with its own (logical) snapshot of the data. If a transaction t writes an object x, a new version of x is created that is visible for transactions that started after t committed. SI is widely applied because of its good performance characteristics, but it permits non-serializable schedules [BBG+95]. If the bank applies SI to schedule transactions t_1 and t_2 this can lead to a violation of constraint C defined over the accounts of Alice and Bob. For instance, schedule s shown in Figure 5.1 is a schedule of t_1 and t_2 that can be produced by SI. SI allows both transactions to commit, which results in a violation of constraint C since after the execution of the requests the combined balance of accounts x and y is less than zero. Note that s is not serializable, i.e., there exists no serial execution of t_1 and t_2 that is equivalent to s.

The SSI implementation follows the approach from Cahill et al. [CRF09]. This approach identifies access patterns in SI schedules that can lead to non-serializability and proactively aborts transactions participating in such patterns. We call this type of pattern *potential pivot structure* (PPS). A schedule s contains a PPS if there are three non-aborted transactions t_i, t_j and t_k (t_i and t_k are not necessarily distinct) so that there is an *rw-dependency* between t_i and t_j, and between t_j and t_k. An *rw-dependency* exists between two concurrent transactions t_i and t_j if t_i read a

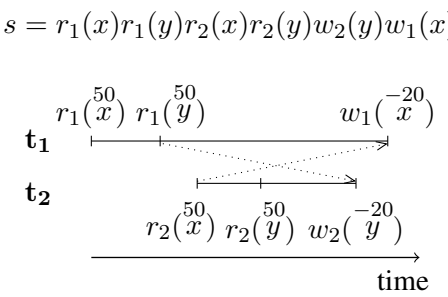


Figure 5.1: Snapshot Isolation History for Banking Example

version of x and t_j wrote a later version of x. For example, the schedule shown in Figure 5.1 contains two rw-dependencies (shown as dotted arrows) that form a PPS between t_1 and t_2. The SSI protocol aborts either transaction t_1 or t_2 to break the PPS. Breaking all PPSs in a schedule guarantees serializability and, thus, no constraint violations can occur.

5.3 Protocol Comparison

ODA provides predefined protocol implementations, such as SI, SSI or SS2PL, as well as predefined workloads including an extended version of the banking example. The user selects a workload and one or more protocols. ODA will then simultaneously execute these protocols on the workload, and create one schedule per protocol. ODA displays the current states of the data and scheduling relations for each scheduling iteration and protocol. This allows for a direct comparison of the protocols, as shown in Figure 5.2.

Example 22. The user debugs SI and SSI by letting ODA schedule the banking workload for both protocols. Figure 5.2 shows the ODA GUI with the scheduling relations for SI and SSI. Figure 5.4 shows the state of relation $\mathcal{H}$ for schedule s from Figure 5.1. Attribute ID models the order of requests in the schedule, TA is the transaction identifier, Seq is the relative position of the request within the transaction, Op denotes the operation type, Ob the object, and Val is the value to be written for write requests. For example, the read request $r_1(x)$, the first request of transaction t_1, is modeled as tuple $(1, 1, 1, r, x, \epsilon)$. Here, ϵ denotes a null value.

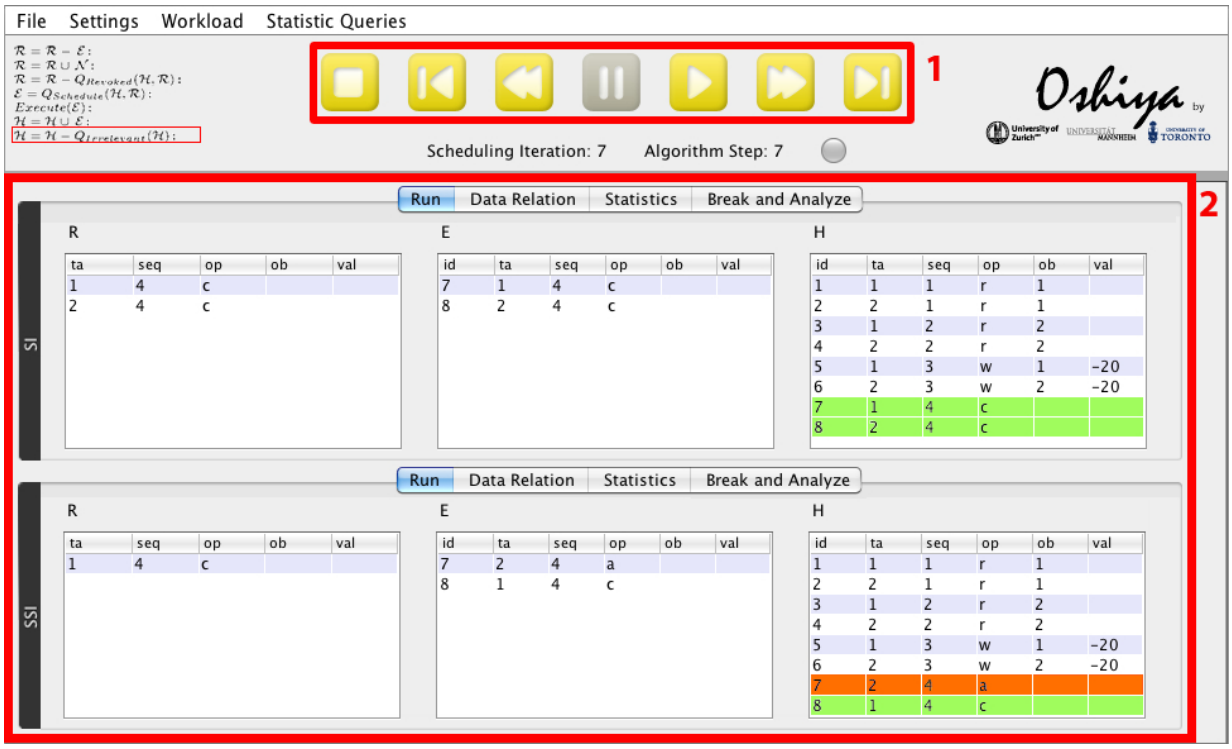


Figure 5.2: Navigational Controls and Relations Capturing Scheduling State

5.4 Break and Analyze Queries

ODA allows setting breakpoints that stop scheduling of requests when a certain event occurs. We specify break points through *break queries* (BQs) that are executed after each scheduling iteration. Break queries are expressed over the scheduling and data relations. Scheduling is stopped when a break query returns a non-empty result. Whenever a break query causes scheduling to stop, its results are presented to the user. Thus, the results of break queries are used to provide contextual information about the event that caused execution to stop.

Example 23. To identify violations of constraint C defined over the accounts from the banking example, the user registers break query BQ_{CV}, given below as a relational calculus expression. BQ_{CV} stops scheduling whenever the sum S of the balances of the accounts of an owner is negative. Subquery $Aggr$ sums the balances B for the accounts of each owner O (aggregation with group-by).

$$BQ_{CV} = \{O, S \mid Aggr(O, S) \wedge S < 0\}$$

$$Aggr = \{O, SUM(B) \mid Accounts(\_, B, \_, O)\}$$

For SI, BQ_{CV} detects the constraint violation for the accounts of owner AB as shown in Figure 5.3. BQ_{CV} always returns an empty set for SSI, because SSI ensures serializability which prevents constraint violations.

BQ_{CV}

Owner	Sum
AB	-40

$Accounts$

Acc	Bal	Type	Owner
v	50	checking	D
w	50	saving	D
x	-20	checking	AB
y	-20	saving	AB

Figure 5.3: Result of Break Query $\mathrm{BQ_{CV}}$

When a break query fires and stops execution, the user typically wants to investigate why execution has stopped. For instance, for each tuple in the result of BQ_{CV}, we want to know the requests in relation $Accounts$ that led to the constraint violation. In ODA, we get this functionality through *analyze queries* (AQs). An analyze query AQ is defined for a break query BQ and a relation R. The purpose of an analyze query is to identify a subset of the tuples of relation R that are related to a certain tuple in the result of BQ. To realize this parameterization with the values of a result tuple, we allow the analyze query to reference attributes from the result schema of BQ through a set of distinguished variables. If execution stopped because a break query BQ returned a non-empty result, the user can select a tuple t from the result of BQ. This triggers the system to execute the analyze queries registered for BQ. Before executing an analyze query, it is parameterized with the values from tuple t. Afterwards, ODA highlights the tuples returned by the analyze query in relation R.

Example 24. To investigate the constraint violations detected by BQ_{CV} for SI, the user registers the following analyze query to retrieve the accounts from relation $Accounts$ that caused the violation:

$$AQ_{CV} = \{A, B, T, O \mid Accounts(A, B, T, O) \wedge O = \$1\}$$

If the user selects tuple $(AB, -40)$ from the result of BQ_{CV} (Figure 5.3), then AQ_{CV} is parameterized with $\$1 = AB$. In this case, AQ_{CV} returns the accounts from relation $Accounts$ that belong to owner AB. These account tuples are highlighted by ODA as shown in Figure 5.3. Thus, the user now knows which accounts caused the constraint violation.

Since ODA models both the scheduling and *data relations* of the backend database as temporal relations, break queries have access to the full transaction-time history of these relations. Break queries are more general than break and watchpoints of program debuggers in that (1) by accessing the state history we support break conditions that are not possible or cumbersome to express with standard debuggers such as "stop after three PPSs have been broken", and (2) with analyze queries we partially automate the common debugging pattern of closely examining the state (values of variables in a program) when execution is stopped.

5.5 Navigational Debugging

ODA provides the user with navigational controls to debug a protocol after execution was stopped by a break query. The navigational controls are shown in Figure 5.2 (mark 1). Besides starting and stopping the scheduler, we support stepping forward and backward one scheduling iteration. These controls allow the user to browse forward and backward through the protocol execution in order to (1) understand how the detected pattern occurred and (2) to see how the protocol handles the pattern.

Example 25. The reason why constraint violations do not occur under SSI is that SSI detects PPSs and aborts one of the participating transactions. In this example, we show how to use navigational controls and break queries to analyze how PPSs are handled by SI and SSI. We register break query BQ_{PPS}, shown below, that stops scheduling whenever a PPS is detected in the schedules of SI or SSI.

$$BQ_{PPS} = \{T_1, T_2, T_3 \mid RW(T_1, T_2) \wedge RW(T_2, T_3)\}$$
$$RW = \{T_1, T_2 \mid \exists O : \mathcal{H}(\_, T_1, \_, r, O, \_) \wedge \mathcal{H}(\_, T_2, \_, w, O, \_) \wedge Conc(T_1, T_2)\}$$

Break query BQ_{PPS} checks for PPSs by identifying two consecutive rw-dependencies between transactions T_1, T_2, and T_3. Subquery RW detects rw-dependencies in $\mathcal{H}$ between the concurrent transactions T_1 and T_2, i.e., T_1 read a version of an object O and T_2 wrote a version of O. We omit the definition of $Conc$, which returns all pairs of concurrently executed, non-aborted transactions from $\mathcal{H}$. An analyze query is registered to highlight all requests in $\mathcal{H}$ that form a PPS identified by BQ_{PPS}.

When executing SI and SSI over the banking workload, break query BQ_{PPS} returns tuples with the identifiers of the three participating transactions for each PPS (see Figure 5.4). The user selects tuple $(1, 2, 1)$ which causes ODA to execute the registered analyze query to highlight the requests participating in the PPS (highlighted tuples in Figure 5.4). We use the navigational controls to step forward to compare how SI and SSI handle the PPS and to check whether the SSI implementation aborts one of the transactions participating in the PPS, which is sufficient to ensure serializable schedules. Stepping forward until the commit requests c_1 and c_2 of transactions t_1 and t_2 have been scheduled, the user observes how SI and SSI handle these requests. This scheduling state is shown in Figure 5.2 (mark 2). SI allows both t_1 and t_2 to commit, i.e., tuples $(7, 1, 4, c, \epsilon, \epsilon)$ and $(8, 2, 4, c, \epsilon, \epsilon)$ are added to relation $\mathcal{H}$ (green highlighted tuples). SSI aborts transaction t_2 by inserting tuple $(7, 2, 4, a, \epsilon, \epsilon)$ into $\mathcal{H}$ (orange highlighted tuple) in order to break the PPS and to prevent constraint violations.

BQ_{PPS}

T1	T2	T3
1	2	1
2	1	2

$\mathcal{H}$

ID	TA	Seq	Op	Ob	Val
1	1	1	r	x	
2	1	2	r	y	
3	2	1	r	x	
4	2	2	r	y	
5	2	3	w	y	-20
6	1	3	w	x	-20

Figure 5.4: Result of Break Query $\mathrm{BQ_{PPS}}$

5.6 Statistical Protocol Analysis

In order to investigate the global properties of a schedule, such as the number of occurrences of a pattern, we need to access information about the current scheduling iteration and aggregated information about the complete history up to the current iteration. ODA automatically keeps default global statistics about a protocol execution, e.g., size of scheduling relation state. Fur-

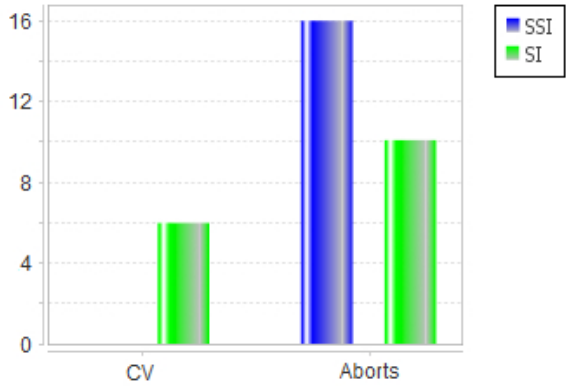


Figure 5.5: Aborts and Constraint Violations for SSI and SI

thermore, ODA allows the user to register new measures as *statistic queries* (SQs) that are run for all protocols in the system after each scheduling iteration. A statistic query is required to produce a single result tuple with a single numerical attribute. Statistic queries can access scheduling relations, data relations, and the results of break queries. This enables the user to reuse break queries for collecting statistics. The result of each statistic query is captured per iteration and aggregated to an average and total value. ODA provides several types of visualization to display the captured statistics, e.g., per iteration in line charts or as total values in bar charts, pie charts or tables.

Example 26. As shown in Section 5.5, in contrast to SI, SSI checks for PPSs and aborts one of the participating transactions. To evaluate the additional aborts of SSI over SI, we register statistic query SQ_{Abort}, which counts the number of aborts in relation $\mathcal{H}$. We measure how many constraint violations were prevented by registering statistic query SQ_{CV} that returns the number of constraint violations.

$$SQ_{Abort} = \{COUNT(T) \mid \mathcal{H}(\_, T, \_, a, \_, \_)\}$$
$$SQ_{CV} = \{COUNT(B) \mid BQ_{CV}(B, \_)\}$$

Figure 5.5 shows a visualization for these statistics when measured over an extended banking example with more accounts and constraints of the same type as constraint C. The results of the statistic queries for both SI and SSI are visualized in a single bar chart.

5.7 Conclusions

ODA is a powerful debugging and analysis tool for Oshiya protocol implementations. It models and extends standard debugging features such as break points in a declarative manner. We illustrate how to debug, analyze and compare protocol implementations with ODA.

Besides using ODA to compare protocols, as shown in this chapter, the main features of the system are also extremely useful when developing new scheduling protocols. For instance, we can register a break query to detect a certain type of error and a statistic query to count the number of occurrences. The statistic query is run after each modification as a type of regression test. The coupling between statistic and break queries enables us to debug the protocol starting from the scheduling iteration where a test failed. Furthermore, after modifying a protocol implementation we can use ODA to compare the old and new version to determine if our modification resulted in the intended behavior.

CHAPTER 6

Smile - Declarative Scheduling Middleware

Abstract

Modern server systems schedule large amounts of concurrent requests constrained by, e.g., correctness criteria and service-level agreements. Since standard database management systems provide only limited consistency levels, the state of the art is to develop schedulers imperatively which is time-consuming and error-prone. In this chapter, we present *Smile* (declarative Scheduling MIddLEware), a tool for developing domain-specific scheduling protocols declaratively. Smile decreases the effort to implement and adapt such protocols because it abstracts from low level scheduling details allowing developers to focus on the protocol implementation. We demonstrate the advantages of our approach by implementing a domain-specific use case protocol.

6.1 Introduction

Modern application servers handle large numbers of concurrent requests which have to be scheduled according to, e.g., correctness criteria like classical serializability or service-level agreements (SLAs). Standard database management systems (DBMSs) offer a limited amount of fixed consistency levels, do not provide sophisticated support for SLAs and, thus, often cannot be used to satisfy domain-specific scheduling requirements. The state of the art is to develop schedulers imperatively for applications like Amazon, Ebay or Yahoo [CRS+08, Vog07] which yields fine-tuned schedulers satisfying the application's scheduling constraints. But procedural implementations of schedulers can be complex and difficult to understand, especially if the request types and correctness criteria are less well studied than, e.g., classic serializability. Adapting schedulers to evolving requirements results in expensive and error-prone re-implementations. With our approach we address these issues by leveraging a declarative language to implement schedulers which has been shown to be beneficial in previous work [ACC+10, Til10].

6.1.1 Banking Scenario

We use the following simplified banking scenario to illustrate the shortcomings of standard DBMSs with regard to non-standard scheduling requirements. A bank institute serves normal and premium customers holding bank accounts. A domain expert defines the following constraints: (C1) Account data has to be accessed under strong consistency to obviate inconsistent states and (C2) Do not schedule requests for normal customers, if there are pending requests from premium customers. How can a scheduler developer implement these constraints? Constraint C1 can be realized with standard DBMSs by applying a high isolation level, but C2 is not supported by standard DBMSs. The alternative is to develop a new scheduler from scratch which is expensive and error-prone.

6.2 Smile: Declarative Scheduling Middleware

Smile, our declarative scheduling middleware prototype, allows the implementation of domain-specific scheduling constraints. Executable scheduling protocols are specified with few lines of code, paving the way for sophisticated and easy-to-reason-about scheduling protocols. Our

approach is based on a generic formal framework called Oshiya[1] that models the scheduling state as a set of so-called *scheduling relations*, e.g., one relation models the schedule produced so far. Scheduling logic is encapsulated in a set of declarative queries called *scheduling queries*. To produce a schedule (sequence of scheduling relation states), Smile schedules multiple requests at the same time by repeatedly executing the scheduling queries over the scheduling relations. See Chapter 2 for an in-depth discussion.

This approach has several advantages: **(1)** Smile abstracts from low level scheduling details that are independent of the scheduling constraints such as parallelism in the scheduler code, queueing of incoming requests, or managing (network) connections. Developers can focus on the protocol implementation (the scheduling queries) itself, which decreases the amount of code and the effort needed to implement or adapt schedulers. We developed scheduling queries for the strong two-phase locking (SS2PL) protocol [TGBK10] as well as for a relaxed consistency protocol (see Chapter 4). **(2)** Smile's underlying model allows to specify scheduling protocols close to their formal definition, facilitating reasoning over properties of protocol implementations such as verifying their correctness. For instance, we have proven the correctness of the scheduling queries implementing SS2PL [TGBK10]. **(3)** The separation of scheduling logic and scheduler implementation opens up interesting optimization opportunities that we plan to investigate in future work. E.g., using specialized execution engines to execute scheduling queries and controlling the trade-off between the time spent for scheduling requests and the time spent to execute them. **(4)** Scheduling sets of requests at the same time can improve the performance for large numbers of concurrent requests [Til10].

6.2.1 Smile Architecture

The Smile prototype implements the Oshiya scheduling model using three threads (*ClientWorker*, *Declarative Scheduler* and *Executor*), all running independently and continuously. The Smile architecture is shown in Figure 6.1 with arrows denoting data flow.

ClientWorker This thread manages client connections. The ClientWorker thread receives new requests from clients, buffers these client requests in a queue and periodically inserts them into $\mathcal{R}$ as batch job (Figure 2.2, step 2).

Declarative Scheduler This thread performs request scheduling by periodically executing

[1] Oshiya refers to the passenger arrangement staff at Japanese train stations who help to fill a train by pushing people onto the train or guiding people to free railway cars.

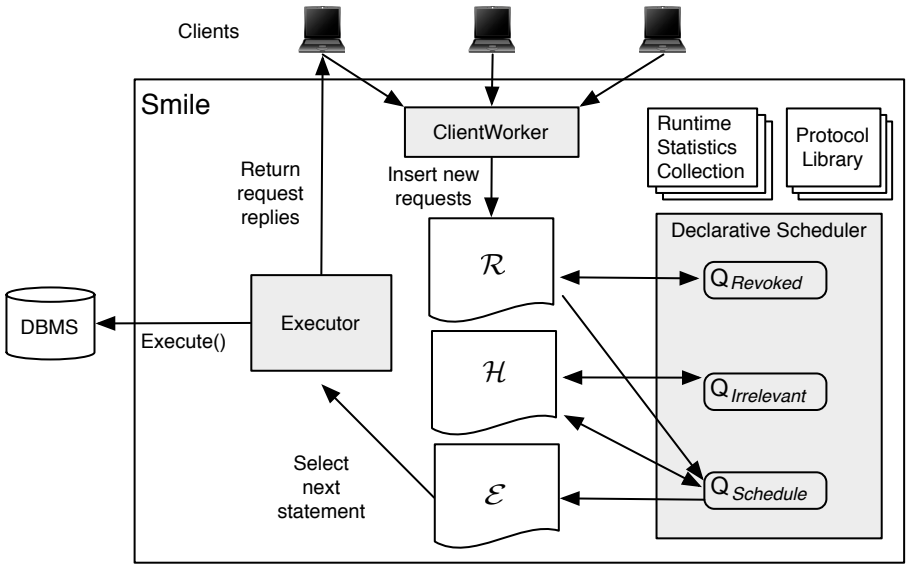


Figure 6.1: Smile Architecture

$Q_{Revoked}$, $Q_{Schedule}$ and $Q_{Irrelevant}$ (Figure 2.2, step 3, 4, 7).

Executor The Executor thread is executing the scheduled requests located in $\mathcal{E}$ against the DBMS by repeating the following steps: Retrieve the request with the smallest *ID* from $\mathcal{E}$, execute it against the back-end DBMS, return the request result to the client that has sent this request, and delete it from $\mathcal{E}$ (Figure 2.2, step 5).

Protocol Library Smile offers a protocol library providing the scheduler developer with pre-cooked scheduling queries (e.g., for SS2PL). These scheduling queries can be used out of the box or as a starting point to develop domain-specific protocols. We expect developers to extend this library over time with their own protocol modules.

Runtime Statistics Collection We let Smile gather statistics about the behaviour of its operations at runtime such as the cardinalities of $\mathcal{R}$, $\mathcal{H}$ and $\mathcal{E}$ and the execution times of the scheduling queries. In future work, we plan to expose this information to the scheduler developer for the use in the scheduling queries and let her provide policies for scheduling the execution of the Smile threads.

We developed strategies deciding when to pause a thread ensuring an efficient resource usage. E.g., running the Executor while $\mathcal{E}$ is empty wastes resources.

6.2.2 Example: Use Case Implementation

We sketch the protocol implementation of the use case to illustrate the simplicity and conciseness of our approach. Scheduling queries are given as domain relational calculus (DRC) expressions. A simplified DRC formulation of $Q_{Schedule}$ implementing the use case constraints is:

$$Q_{Schedule} = \{S, C \mid is2PL(S, C) \wedge (is2PL(\_, premium) \Rightarrow C = premium)\}$$

We use a declarative implementation of SS2PL to realize constraint C1. Predicate $is2PL(S, C)$ uses $\mathcal{R}$ to determine all requests S with their customer class C that can be executed without violating the SS2PL constraints that have to hold for the generated schedule (requests in relation $\mathcal{H}$). We use S as a shorthand for the request related attributes of $is2PL$ (transaction ID etc.). Using Oshiya, we can implement scheduling constraint C2 as follows: If there exists at least one request of a premium customer ($is2PL(\_, premium)$), then only premium requests are selected by $Q_{Schedule}$ ($C = premium$).

CHAPTER 7

Conclusions and Future Work

In this thesis, we propose the Oshiya scheduling model as a new method for the development of domain-specific protocols for scheduling database requests. This generic model treats requests as data, stores the scheduling state in a database, implements scheduling protocols as declarative queries, and employs database query processing techniques to produce request schedules. Oshiya is capable of expressing traditional and domain-specific scheduling protocols, and provides support for SLAs such as a differentiation between transactions, e.g., transactions from users with different priorities.

We introduce the declarative two-phase locking protocol (DSS2PL), an Oshiya implementation of the traditional strong two-phase locking protocol. Oshiya allowed us to concisely implement DSS2PL with twelve lines of DRC expressions. Whereas, a procedural protocol implementation is difficult to verify, the DSS2PL implementation is close to its formal definition which enabled us to prove its correctness. Our experimental results show that for large numbers of concurrent requests our approach provides better performance and predictability than a native database scheduler.

We develop the Declarative Serializable Snapshot Isolation protocol (DSSI), a modified version of the Snapshot Isolation protocol. We formalize DSSI as an Oshiya protocol specification and

develop an SQL implementation of DSSI using Oshiya. Our implementation is concise and close to the formal protocol specification. This allowed us to prove that, in contrast to SI, DSSI ensures that every produced schedule is serializable. Our approach requires no analysis of application programs or changes to the underlying DBMS. DSSI provides database independence and can be run on DBMSs that do not support snapshots.

We propose the resource acquisition protocol (RAP), a domain-specific protocol for scheduling concurrent transactions that compete for resources, a typical usage pattern in booking, reservation, and web shop systems. We prove that RAP does not produce aborts due to deadlocks. We introduce the acquisition graph that we use to compare RAP with SS2PL and SI. Our experimental results confirm that RAP performs better than these two protocols with respect to aborts and throughput.

The Oshiya Debugger and Analyzer (ODA) is our system for debugging, visualizing, and comparing scheduling protocols developed using the Oshiya scheduling model. ODA supports the simultaneous execution of single- and multiversion protocols, breakpoints, backward and forward debugging, as well as the statistical and visual analysis of protocols.

Future Work We currently limit the types of requests that have to be scheduled to atomic database requests (read/write/abort/commit). The decision to use only these requests is based on the fact that web applications typically manipulate one record at a time [CRS+08]. For example, most of the Amazon services store and retrieve data by primary key only (requests specify the primary key) and do not require complex querying and full DBMS functionality [DHJ+07, Vog07]. It is interesting future work to further investigate this issue and extend Oshiya to support more complex queries like range queries. For example, Oshiya could be extended to logically lock ranges of keys, similar to [LM09].

We further plan to implement Oshiya in the kernel of the PostgreSQL open source database management system. Oshiya can be integrated in a way that it substitutes the native scheduler allowing to add and modify protocols in a flexible manner. To improve the scalability of this DBMS, Oshiya could also be added in addition to the existing scheduler. Once scheduling request per request encounters scalability problems, the Oshiya scheduler can take over request scheduling applying its set-at-a-time scheduling approach.

We will also investigate techniques to improve RAP in order to reduce the number of availability aborts, i.e., aborts due to an insufficient resource availability. One promising approach is to return fake quantities of zero if the quantity of a resource falls below a threshold. This would

cause clients to fail during browsing if the probability of a later availability abort is high, i.e., we avoid long-running transactions that are bound to fail at the cost of potentially unnecessary aborts.

Bibliography

[AABM08] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Modeling and Exploiting Query Interactions in Database Systems. In *Proceeding of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 183–192, New York, NY, USA, 2008. ACM.

[ACC+10] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 223–236, New York, NY, USA, 2010. ACM.

[AFR09] Mohammad Alomari, Alan Fekete, and Uwe Röhm. A Robust Technique to Ensure Serializable Executions with Snapshot Isolation DBMS. In *Proceedings of the 25th International Conference on Data Engineering*, ICDE '09, pages 341–352, Washington, DC, USA, 2009. IEEE Computer Society.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[BF99] N. Bhatti and R. Friedrich. Web server support for tiered services. *Network, IEEE*, 13(5):64–71, Sep/Oct 1999.

[BFG+06] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed-Currency Serializability for Middle-Tier Caching and Replication. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 599–610, New York, NY, USA, 2006. ACM.

[BFG+08] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a Database on S3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 251–264, New York, NY, USA, 2008. ACM.

[BMK08] Alexander Böhm, Erich Marth, and Carl-Christian Kanne. The Demaq System: Declarative Development of Distributed Applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1311–1314, New York, NY, USA, 2008. ACM.

[CdMP08] Angelo Chianese, Antonio d'Acierno, Vincenzo Moscato, and Antonio Picariello. Pre-serialization of long running transactions to improve concurrency in mobile environments. In *Proceedings of the 24th International Conference on Data Engineering Workshops*, ICDE '08 Workshops, pages 129–136. IEEE Computer Society, 2008.

[CMZ04] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepole. C-JDBC: Flexible Database Clustering Middleware. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 9–18, Berkeley, CA, USA, 2004. USENIX Association.

[CPT+07] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The Design and Implementation of a Declarative Sensor Network System. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 175–188, New York, NY, USA, 2007. ACM.

[CR90] Panayiotis K. Chrysanthis and Krithi Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceed-*

ings of the 1990 ACM SIGMOD international conference on Management of data, SIGMOD '90, pages 194–203, New York, NY, USA, 1990. ACM.

[CRF09] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.*, 34(4):1–42, 2009.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[DB2] DB2 Query Patroller. http://www-01.ibm.com/software/de/data/db2ims/db2ud.html.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[ENTZ04] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 276–286, New York, NY, USA, 2004. ACM.

[FBJ09] Shel Finkelstein, Rainer Brendle, and Dean Jacobs. Principles for Inconsistency. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*, CIDR '09. www.crdrdb.org, 2009.

[Fek99] Alan David Fekete. Serializability and Snapshot Isolation. In *Proceedings of the 10th Australasian Database Conference*, volume 21 of *ADC '99*, pages 201–210. Springer, 1999.

[Fek05] Alan Fekete. Allocating Isolation Levels to Transactions. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, pages 206–215, New York, NY, USA, 2005. ACM.

[FK09] Daniela Florescu and Donald Kossmann. Rethinking Cost and Performance of Database Systems. *SIGMOD Rec.*, 38(1):43–48, 2009.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[GDJ+11] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI '11, pages 239–252, Berkeley, CA, USA, 2011. USENIX Association.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, pages 249–259, New York, NY, USA, 1987. ACM.

[HC09] Pat Helland and Dave Campbell. Building on Quicksand. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*, CIDR '09. www.crdrdb.org, 2009.

[HD91] Ugur Halici and Asuman Dogac. An Optimistic Locking Technique For Concurrency Control in Distributed Databases. *IEEE Trans. Software Eng.*, 17(7):712–724, 1991.

[JFRS07] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the Detection of Snapshot Isolation Anomalies. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1263–1274. VLDB Endowment, 2007.

[KE06] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 6. Auflage*. Oldenbourg, 2006.

[KGR+10] Lucja Kot, Nitin Gupta, Sudip Roy, Johannes Gehrke, and Christoph Koch. Beyond Isolation: Research Opportunities in Declarative Data-Driven Coordination. *SIGMOD Rec.*, 39(1):27–32, September 2010.

[KHAK09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, 2009.

[KKDK07] Stefan Krompass, Harumi Kuno, Umeshwar Dayal, and Alfons Kemper. Dynamic Workload Management for Very Large Data Warehouses: Juggling Feathers and Bowling Balls. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1105–1115. VLDB Endowment, 2007.

[LM09] David Lomet and Mohamed F. Mokbel. Locking Key Ranges with Unbundled Transaction Services. *Proc. VLDB Endow.*, 2(1):265–276, August 2009.

[PA04] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[Sch06a] Schroeder, Bianca and Harchol-Balter, Mor and Iyengar, Arun and Nahum, Erich M. Achieving Class-Based QoS for Transactional Workloads. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 153. IEEE Computer Society, April 2006.

[Sch06b] Schroeder, Bianca and Harchol-Balter, Mor and Iyengar, Arun and Nahum, Erich M. and Wierman, Adam. How to Determine a Good Multi-Programming Level for External Scheduling. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 60. IEEE Computer Society, April 2006.

[SLSV95] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.*, 20:325–363, September 1995.

[SS03] André Seifert and Marc H. Scholl. Processing Read-Only Transactions in Hybrid Data Delivery Environments with Consistency and Currency Guarantees. *Mob. Netw. Appl.*, 8(4):327–342, 2003.

[TGBK10] Christian Tilgner, Boris Glavic, Michael H. Böhlen, and Carl-Christian Kanne. Correctness Proof of the Declarative SS2PL Protocol Implementation. Technical Report IFI-2010.0008, University of Zurich, Department of Informatics, Zürich, Switzerland, September 2010.

[TGBK11a] Christian Tilgner, Boris Glavic, Michael H. Böhlen, and Carl-Christian Kanne. Declarative Serializable Snapshot Isolation. In *Proceedings of the 15th East Euro-*

pean Conference on Advances in Databases and Information Systems, ADBIS '11, pages 170–184. Springer, 2011.

[TGBK11b] Christian Tilgner, Boris Glavic, Michael H. Böhlen, and Carl-Christian Kanne. Smile: Enabling Easy and Fast Development of Domain-Specific Scheduling Protocols. In *Proceedings of the 28th British National Conference on Databases*, BNCOD '11, pages 128–131. Springer, 2011.

[Til10] Christian Tilgner. Declarative Scheduling in Highly Scalable Systems. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, pages 41:1–41:6, New York, NY, USA, 2010. ACM.

[Vog07] Werner Vogels. Data Access Patterns in The Amazon.com Technology Platform. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, page 1. VLDB Endowment, 2007.

[WDK+07] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling Games to Epic Proportions. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 31–42, New York, NY, USA, 2007. ACM.

[WV02] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[YG09] Aravind Yalamanchi and Dieter Gawlick. Compensation-Aware Data Types in RDBMS. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 931–938, New York, NY, USA, 2009. ACM.

[YSRG06] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A High-Level Language for Data-DrivenWeb Applications. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 32, Washington, DC, USA, 2006. IEEE Computer Society.

Curriculum Vitae

Personal Details

Name:	Christian Michael Tilgner
Date of Birth:	March 16, 1981
Citizenship:	German

Education

2006-2012	Doctoral student Database Technology Group (DBTG) Department of Informatics University of Zurich
2006	Master of Science in Computer Science University of Leipzig, Germany
2004-2006	Master student in Computer Science University of Leipzig, Germany
2004	Diplom-Informatiker (FH) University of Applied Sciences Deutsche Telekom AG
2000-2004	Diploma student in Computer Science and Telecommunication University of Applied Sciences Deutsche Telekom AG Leipzig, Germany